INTRODUCING SPROCKET – OUR NEW TINY FRAMEWORK!

MacTech MAGAZINE™

Formerly MacTutor

FOR MACINTOSH PROGRAMMERS & DEVELOPERS

OCTOBER 1994 / VOLUME 10, NO. 10

n This ssue!

10
32128 74887 8

$5.85 US
$6.95 Canada
ISSN 1067-8360
Printed in U.S.A.

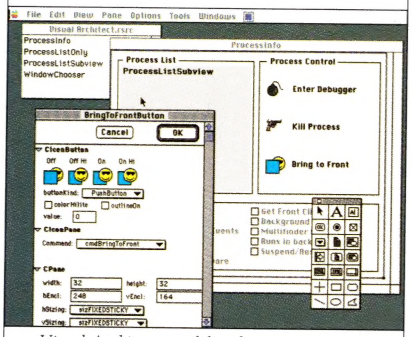
We've got something to show you that you will have to see with your own eyes to believe.

It's Symantec C++ 7.0. And if you're a Macintosh developer, you'll want to take a good look.

VISUAL ARCHITECT. THE EASIEST WAY TO DESIGN A MACINTOSH INTERFACE.

Our new Visual Architect lets you visually design and create all of your windows, dialogs, bitmaps, alerts, controls and menus. Then it automatically generates your complete Mac application.

You can test the user interaction of your Macintosh application immediately while in Visual Architect's prototype mode, without even having to compile your application.



Visual Architect simplifies the creation of your Macintosh interface so you can focus on the big picture: creating killer applications.

Completely integrated within our THINK Project Manager environment, Visual Architect is immediately accessible to you every step of the way. Simplifying your entire development process from design to the final code.

To help cut your development cycle even further, we've also included a brand new object browser that more efficiently lists, examines and modifies all objects instantiated during the course of your applications development.

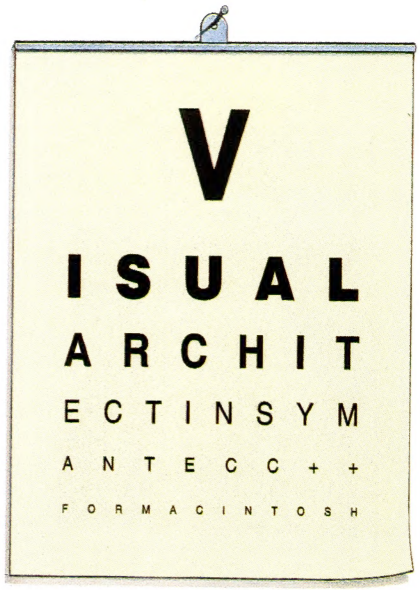
We've also enhanced our THINK Class Library 2.0 (TCL 2.0) with C++ pointers and memory management, new support for AppleEvents and scriptable apps, exception handling support and of course, object-persistence.

THE PATH TO POWER MACINTOSH.

Our new TCL 2.0 uses universal headers, removes all the assembly code specific to 68K and it also supports Power Macintosh procedural pointers.

So all of your TCL applications written in Symantec C++ 7.0 will port easily to Power Macintosh giving you a head start on your Power Macintosh development.

What's more, we're offering you a developers' pre-production release of our Power Macintosh Cross Development Kit. It has the tools you need to get started on creating native Power Macintosh applications today, including a copy of Apple's Power Macintosh linker and debugger.



Any way you look at it, Symantec C++ 7.0 for Macintosh with its Visual Architect is the easiest way to design the Macintosh user interface.

NEW SYMANTEC C++ 7.0. YOU'D HAVE TO BE BLIND TO PROGRAM MACINTOSH WITH ANYTHING ELSE.

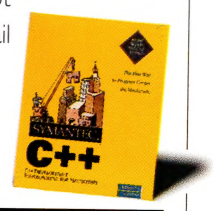
It's a way to speed up the development of your Macintosh applications right now – and be the first to move your programs to the Power Macintosh platform.

CALL 1-800-628-4777.

Ask for ext. 9H16 to upgrade to Symantec C++ 7.0 for Macintosh today for just $149.95* and get our developers' pre-production release of the Power Macintosh Cross Development Kit for just $100.

This offer is not available in any retail store. So call now.

And get your code ready for Power Macintosh.



SYMANTEC.®

*$499.00 Suggested Retail Price. Power Macintosh Development Kit available only with purchase of or upgrade to Symantec C++ 7.0. For more information in Canada, call 1-800-667-8661.

Time Is On Your Side

Now VIP-C creates applications that are accelerated for Power Macintosh.

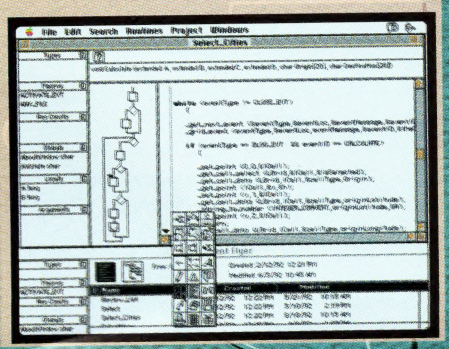
Now, create 68K and Power Macintosh applications faster than ever before. VIP-C's visual design, its application building aids, and its ability to let you modify your program code and interface with immediate feedback, give you a real programming edge. Read further to see why, with VIP-C, time really is on your side...

Complete Development System

Now develop with just one comprehensive system. VIP-C integrates all the tools you need to create stand-alone 68K and Power Macintosh applications—right out of the box! Develop applications in a standard language using this intelligent, full-featured Rapid Application Development (RAD) environment. No more moving from tool to tool. No more dependence on non-standard languages.

For All Levels of Programmers

VIP-C offers multiple levels of support to best fit your abilities. If you just want to type C code into its editor, you can—and VIP-C will automatically check the syntax and create a flowchart of your code. For higher-level support, VIP-C features prewritten intelligent prototypes of all the Mac Toolbox calls, high-level VIP-C Functions that simplify low-level Toolbox calls, Resource Editors that let you visually design and create your windows, menus, dialogs, buttons, etc., and even a VIP-C Dispatcher to provide an efficient and reliable application framework—all in one complete development environment.



User Interface Generation Tools

The VIP-C Dispatcher simplifies program development by automating the main event loop. It acts as a central controller to manage program events by distributing tasks to different routines. User interface items—menus, dialogs, buttons, etc.—are created with integrated resource editors that automatically link user action to your code.

VIP-C 1.5
Everything you need to create complete Macintosh and Power Macintosh applications in one box.

Powerful Prewritten Functions

Functions for the most frequently used features of the Mac interface have been prewritten and stored in a palette for instant use. With the click of a mouse you can lighten your programming workload and speed design and coding time by several orders of magitude.

On-Line Macintosh Toolbox Calls

VIP-C accesses the complete Mac Toolbox. Search and display prototypes for any function, structure, or macro. No more rooting through multiple volumes of Inside Macintosh!

Attention Database Developers!

VIP-C users now have the power to create commercial quality, multi-user, relational databases using the new VIP Database Manager (sold separately). VIP Database Manager has the built-in functionality of the C-Index Pro database engine (recently chosen by Apple OSA for creating the eWorld content publishing software) giving you an inexpensive way to create professional databases, royalty-free!

Compiler Hotlink

If you have a favorite complier, use it! Compile and run your application directly from a menu item in the VIP-C editor. VIP-C supports: THINK C, MPW-C, and CodeWarrior (each sold separately).

Buy VIP-C now for a Limited-Time Special Price of only $295. Act now and get VIP Database Manager for only $149.

VIP-C's suggested retail price is $495. But from now until Oct. 31, 1994, you can order VIP-C for $295. Call Mainstay to order today.

Mainstay

(805) 484-9400

591-A Constitution Ave.
Camarillo, CA 93012
fax: (805) 484-9428

71 rue des Aïrebates
B-1040 Brussels, Belgium
32-2/733.97.91

MacTech Magazine Formerly MacTutor
FOR MACINTOSH PROGRAMMERS & DEVELOPERS
A publication of XPLAIN CORPORATION

How To Communicate With Us

In this electronic age, the art of communication has become both easier and more complicated. Is it any surprise that we prefer **e-mail?** If you have any questions, feel free to call us at 310/575-4343 or fax us at 310/575-0925.

DEPARTMENTS	Internet	CompuServe	AppleLink	America Online	GEnie
Orders, Circulation, & Customer Service	custservice@xplain.com	71333,1063	MT.CUSTSVC	MT CUSTSVC	MACTECHMAG
Editorial	editorial@xplain.com	71333,1065	MT.EDITORIAL	MT EDITORS	–
Programmer's Challenge	progchallenge@xplain.com	71552,174	MT.PROGCHAL	MT PRGCHAL	–
Ad Sales	adsales@xplain.com	71552,172	MT.ADSALES	MT ADSALES	–
Accounting	accounting@xplain.com	–	–	–	–
Marketing	marketing@xplain.com	–	–	–	–
Press Releases	pressreleases@xplain.com	–	–	–	–
General	info@xplain.com	71333,1064	MACTECHMAG	MacTechMag	–
Online support area	ftp://ftp.netcom.com/pub/xplain	*type* GO MACTECHMAG	*see* Third Parties: Third Parties (H-O)	*use keyword:* MACTECHMAG	–

AUTHORS & REGULAR CONTRIBUTORS

MacTech Magazine is grateful to the following individuals who contribute on a regular basis. We encourage others to share the technology. We are dedicated to the distribution of useful programming information without regard to Apple's developer status. For information on submitting articles, ask us for our **writer's kit** which includes the terms and conditions upon which we publish articles.

Richard Clark
General Magic
Mountain View, CA

Chris Espinosa
Apple Computer, Inc.
Cupertino, CA

Jörg Langowski
Jörg's Folder
Grenoble Cedex, France

David R. Mark
M/MAC
Arlington, VA

Jordan Mattson
Apple Computer, Inc.
Cupertino, CA

Mike Scanlin
Programmer's Challenge
Mountain View, CA

Symantec Technical Support Group
THINK Division
Eugene, OR

MacTech Magazine (ISSN: 1067-8360 / USPS: 010-227) is published monthly by Xplain Corporation, 1617 Pontius Avenue, 2nd Floor, Los Angeles, CA 90025-9555. Voice: 310/575-4343, FAX: 310/575-0925. Domestic subscription rates are $47.00 per year. Canadian subscriptions are $59.00 per year. All other international subscriptions are $97.00 per year. Domestic source code disk subscriptions are $77 per year. All international disk subscriptions are $97.00 a year. Please remit in U.S. funds only. Second Class postage is paid at Los Angeles, CA and at additional mailing office.

POSTMASTER: Send address changes to **MacTech Magazine**, P.O. Box 250055, Los Angeles, CA 90025-9555.

TABLE OF CONTENTS

October 1994 • Volume 10, No. 10

MacTech MAGAZINE™ Formerly MacTutor
FOR MACINTOSH PROGRAMMERS & DEVELOPERS

THE EDITOR'S PAGE

By Scott T Boyd, Editor

What We Did On Our Summer Vacation in Boston

Well, so it wasn't really a vacation. I wouldn't wish setting up a booth in the World Trade Center without any air conditioning on anyone, especially as a vacation activity. *[You have to understand, exhibitors aren't really considered "people" and therefore don't need air conditioning. - Ed. nst]*

Actually, there was a good bit of fun. Strangely, though, the biggest parties aren't always the best bet. This year at least one of the biggest almost completely succeeded in what must have been a deliberate attempt to screen out the t-shirt crowd. All that effort to have a museum full of suits? Go figure. Two much smaller parties made their mark. Mac the Knife hosted a nice little get-together. The attendee list spoke volumes about the quality of the Knife's sources. The best party, though, had to be one where the party guests supported the party by buying t-shirts. Yes, it was the "System 7.5 Sucks Less" party, and the quality of the production bears witness to the kind of results you can get with a small team of hard-working, dedicated team members. Some good (non-engineering) folks at Apple had a hard time believing that "sucks less" was a good thing, but us programmer types enjoyed the good-natured ribbing. Strange, but there were a lot more t-shirts than suits at this party!

The show itself held a few surprises, and a lot of non-surprises. If there was a theme, it might have been "More of the same, only native!"

The best part of the show for us was the opportunity to spend time with a whole bunch of the people who build the developer tools we use. MacTech Magazine had two booths, one on the floor of the World Trade Center, and a tiny one in the Apple tent. The tent spot was right in the midst of a number of developers, right between Symantec and Metrowerks. During lulls, developer tool authors mingled, swapped stories, and gave each other a hard time. I got to watch as a worker on one side of us talked a guy from the other side out of a t-shirt, saying, "I promise I'll wear it to work." That's something we'd like to see.

Random Show Observations

Much of what we saw in the way of developer tools has already shown up in print here, or will soon. On the other hand, some of what we saw on the show floor gave us some things to think about. For example, RAMDoubler's success evoked exclamations like, "I sure would like to have a big hit and get rich!" from even the most modest of developers. Berkeley Systems had people standing five deep to get free inflatable goodies and watch fun screen "savers". And, although WordPerfect was giving away umbrellas, we saw people standing in line not knowing about the freebies. They wanted to watch WordPerfect's demo because, "There's no way I'm going to put OLE on my Macintosh just to run the big, new version of Word. I want to see what I'm going to be using next" (we don't make this stuff up).

Dayna was showing a cool demo of their wireless networking. Apple was showcasing just how many applications have gone native (although they didn't bother to point out just how many of them were done with CodeWarrior). Computer Chronicles, the PBS TV show, was taping segments all over the show floor. The World Trade Center food was pretty good, and not expensive, either. We expected airport quality and prices.

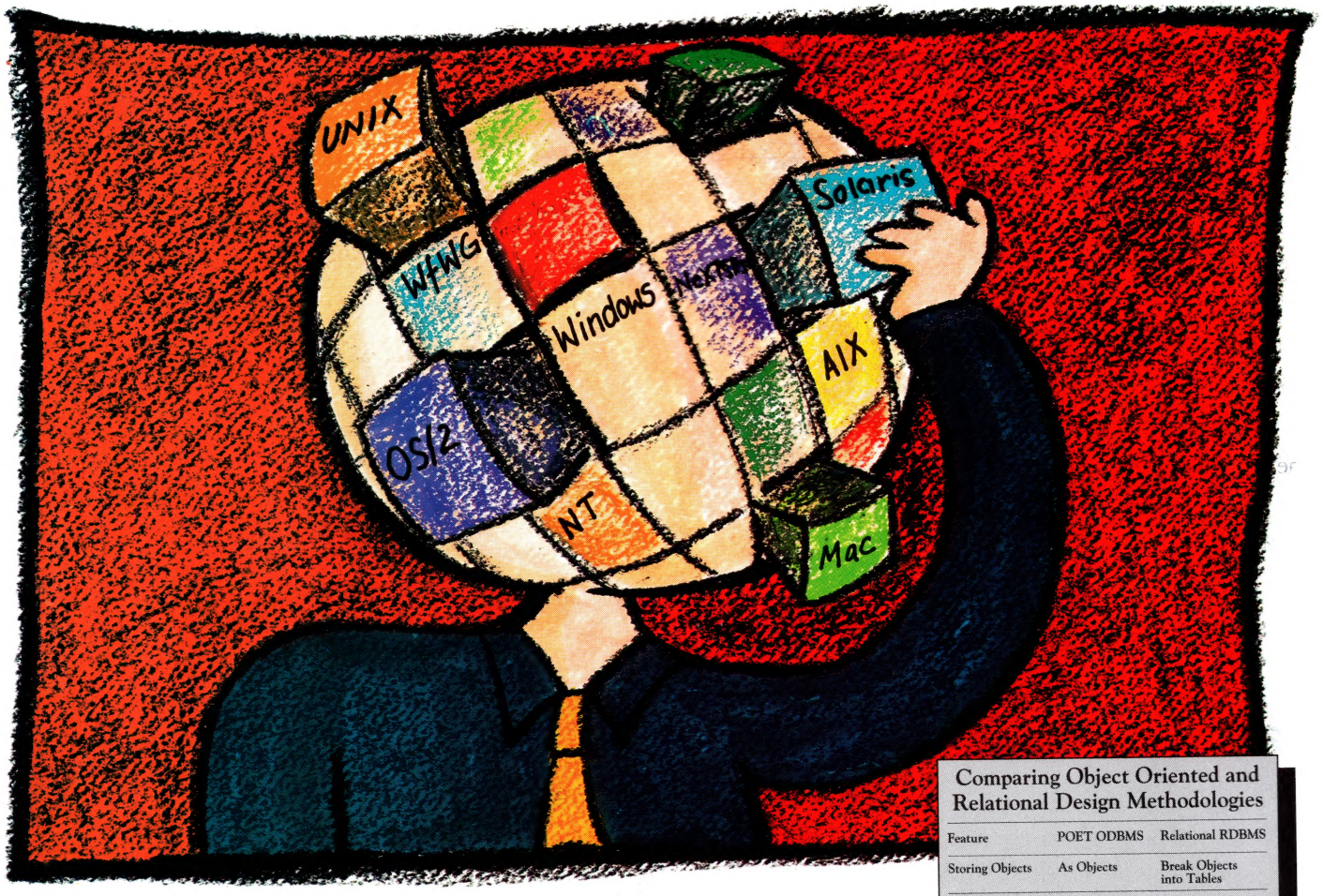
The one game that programmers kept talking about was Sensory Overload, from Reality Bytes. Everyone says it's just like Doom. I (shamelessly) talked them out of a copy so we could do our duty and give you a quick review. In a nutshell, one programmer didn't understand the attraction. Another programmer disappeared for several hours, then cursed me for letting him try it (probably because he couldn't find anywhere to buy it yet).

Even though it was available before the show, going on the road gave Neil a chance to give his new 19.2 PowerPort modem a real workout. He'd been struggling with Apple's modem "solution" for his Duo, and is pleased to report that he's found the alternative. Neil does more e-mail or faxes per minute than anyone I know, and he grooved on Global Village's performance and fax software. He says, "Faxing is now so much faster and easier, it isn't worth comparing to Apple's software. Check it out!"

Speaking of computers and phones, Collaboration Technologies was showing their still-in-development PhoneBridge® to everyone with telephony products, and kicking up quite a stir. PhoneBridge is a hardware/software combo which connects to your Mac via ADB, Sound In, and Sound Out. It can mix and match audio and knows all about phones. You can use it and your Mac as a most interesting phone. The best part? It's a developer platform. I told a few friends about it, and each one immediately went and demanded that Collaboration take their money and give them a developer kit. It was great to see developers truly excited about a new technology! My favorite developer opportunity for it? To use it to mix in the background sound of your choice to create custom atmosphere for your calls (e.g. "Wow, this connection is really bad. I'll have to call you back"). One developer is already working on a real-time Pretty Good Privacy (PGP) secure phone application for it. For more info, e-mail phonebridge@apple.com.

Continued on page 82

Don't even think about using a Relational Database System !



Comparing Object Oriented and Relational Design Methodologies

Feature	POET ODBMS	Relational RDBMS
Storing Objects	As Objects	Break Objects into Tables
Database Model	User Application Model	Separate Database Model Required
C++ Integration	Total	Poor
Database Operations	At Object Level	Must Write Code
Productivity	Increased	Reduced
Complex Object Performance	Excellent	Poor

Use the POET Object Database for C++

C++ and class libraries have made the GUI development much easier. After all, it is only logical that more and more developers think in objects. But object orientation shouldn't end at the user interface programming level.

The Problem: Without POET, a C++ programmer must use flat files or a RDBMS to store objects. He has to write code to overcome the mismatch between the application and the database model. This leads to design restrictions, performance penalties and more code to write and maintain.

The Solution: POET operates at the object level, it speeds up the development process and provides greater performance. Furthermore, the developer can simply take the object-oriented application design in C++ and map it 1 to 1 into the database. Without any compromise at all!

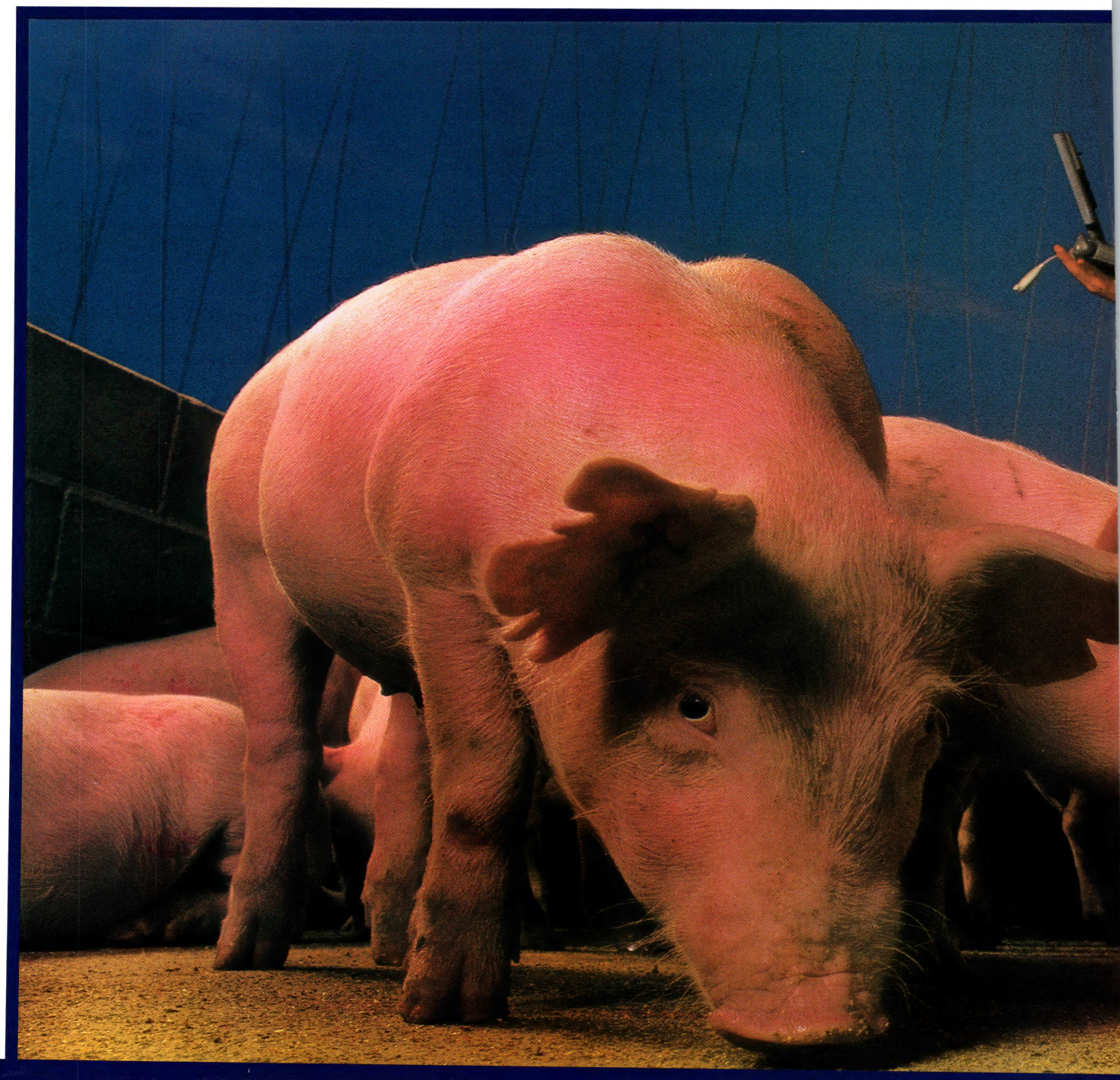
Full featured database: POET provides complete support for Encapsulation, Inheritance, Polymorphism and Containers as well as queries, object locking, and transaction handling .

True cross platform support: Complete interoperability makes the development of network enabled applications a breeze. POET supports Macintosh, Power Macintosh, Windows 3.1, Windows NT, Win32s, Windows for Workgroups, Novell Netware (NLM), SUN, AIX, HP-UX, SGI, SCO, OS/2 and NeXTStep.

Call 1-800-950-8845

POET Software

POET Software Corporation, 4633 Old Ironsides Dr., Suite 110, Santa Clara, CA 95054, Tel.: (408) 970-4640 Fax: (408) 970-4630
United Kingdom (Silicon River) +44 81-316-7777, Germany (POET Software) +49 40-60-99-00, Australia (Microway) +61-3-580-1333, France (LCI) +33-1-34-65-7777
Netherlands (Protools Software) +31 20 645 5023, Sweden (Duke Systems) +46 8 703 2781



"Cozzi Ranch relies on our software to get more pigs to market. To fatten up our profits, we rely on the protection leader."

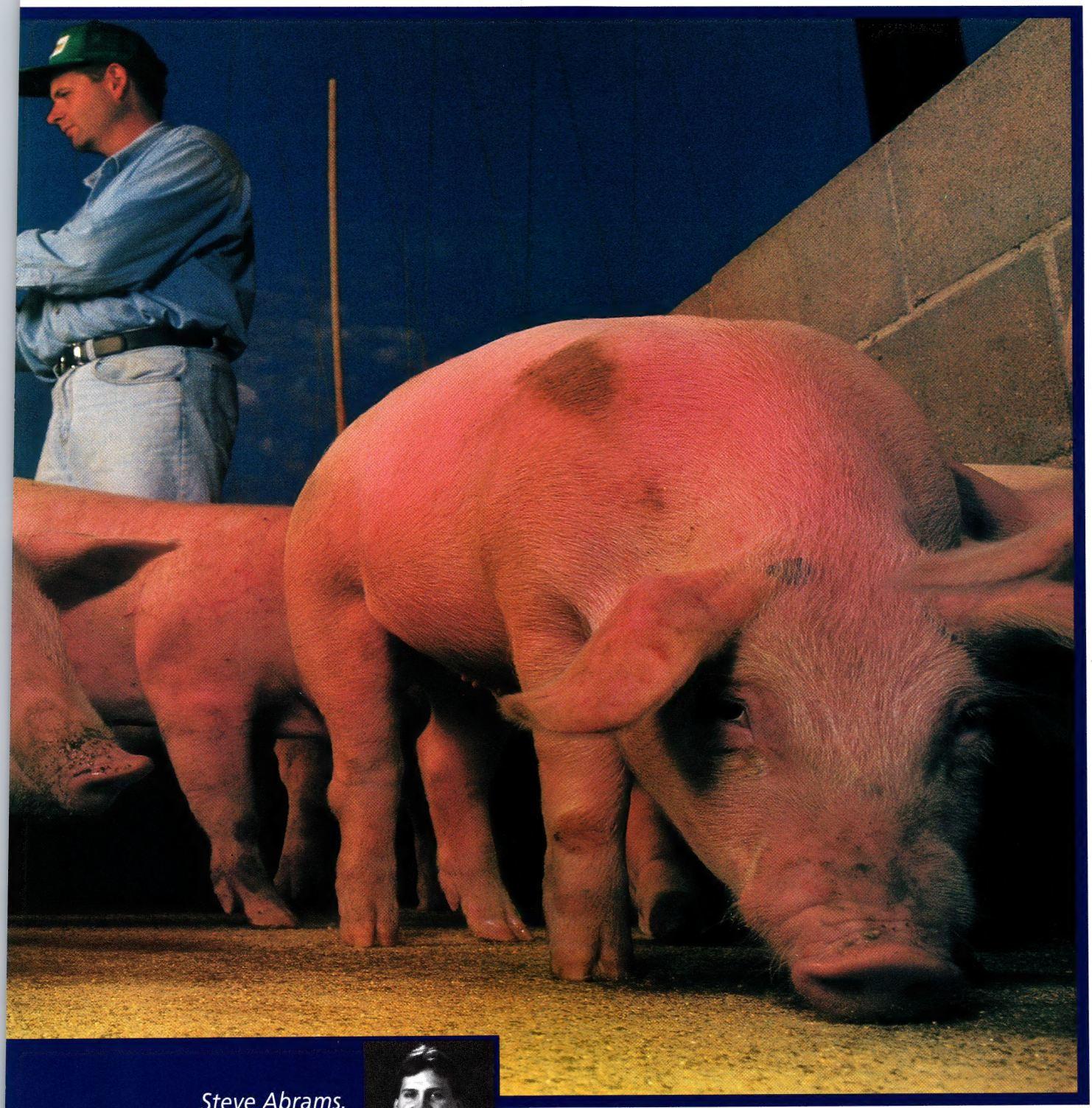
Nothing can eat away profits like a herd of pigs. That's why farmers in 47 countries manage swine production with PigChamp.® And that's why PigChamp protects their revenues with Sentinel,™ from Rainbow Technologies.

As a developer, you know the importance of profit margins. If you sell software, you're probably losing revenue to piracy. Protect with Sentinel, and get the revenue you deserve.

"Our clients produce everything from footballs to kielbasa, and they all get better software at a lower price because we protect with Sentinel.

"Illegal duplication can drive the cost of software sky high. Sentinel lets us sell our products inexpensively worldwide, while serving our customers better," explains PigChamp's Steve Abrams.

RAINBOW TECHNOLOGIES Rainbow Worldwide HQ: Irvine, CA. Phone: (714) 454- 2100 • Fax (714) 454 -8557 • AppleLink D3058 • Rainbow U.K. 44 932 570066 • Rainbow France 33 1 47 38 21 21 • Rainbow Germany 49 89 3217 98 0



Steve Abrams,
PigChamp Software

Over 11,000 developers protect their software with Sentinel. For Macintosh, DOS, Windows, NT, OS/2, UNIX, or any platform – Sentinel is the most advanced protection available. It is the worldwide standard in software protection.

Getting started with Sentinel is quick and easy with a variety of installation options. What's more, Sentinel is truly transparent to your end users. Once installed, they'll never notice it again.

Only Sentinel meets the industry's toughest quality standards and is supported by the industry's largest technical and R&D staff.

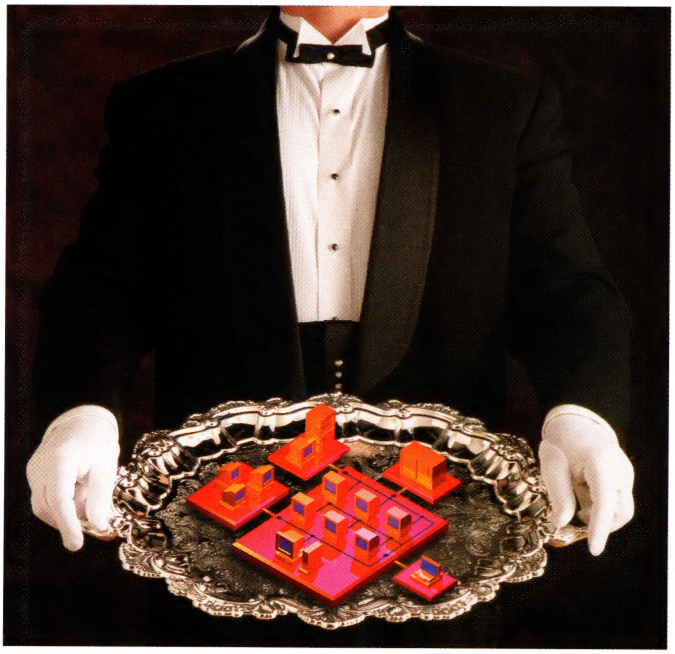
So, watch your profits go hog wild. Protect your revenue with confidence. Protect your software with Sentinel.
Order your Sentinel Developer's Kit today!

1-800-852-8569

SENTINEL™
Securing the future of software.

File-Server.

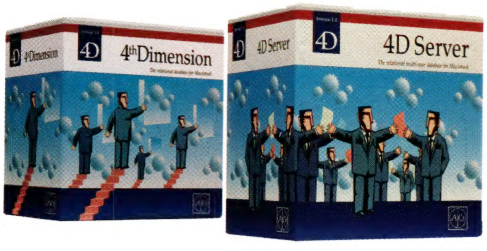
Client-Server.



4D Server delivers on the promises of others.

Recently some software companies have demonstrated "new" versions of their desktop relational databases with the usual claims of incredible performance. What they don't tell you is that their multi-user capabilities are all still based on the same old file-sharing technology. If you add more than a few users, you might as well go back to pen and paper.

Over one year ago ACI introduced 4D Server, a fully integrated high performance client/server database designed from the ground up to give you the performance of large systems costing much more, yet preserve the end-user elegance of our award winning database: 4th Dimension. The result is nothing short of revolutionary.



State-of-the-art performance to all.

Unlike file-sharing systems where only the data itself is on the file server, the client/server architecture of 4D Server means there is a high performance data engine running on the server. Clients send requests to the server. The server processes the requests and sends only the needed information back to the clients.

This division of labor between client and server significantly decreases network traffic and takes maximum advantage of the server hardware. The result is a dramatic increase in speed.

The ease of administration.

4D Server does not require a dedicated database administrator. In fact, all activity is displayed on just one simple screen. It actually takes less time to manage than a traditional file server, not to mention other database servers. Using its built-in multi-tasking system, 4D Server automatically adjusts itself for optimum performance.

Collaborative, friendly development.

4D Server also provides a multi-developer environment. This means several users are able to develop on a single database at the same time, even when the database is in use. 4D Server automatically locks database objects such as layouts and procedures when they are being modified by a database designer, eliminating the usual headache of version control found with single-user development environments.

Knowledge is power, call today!

Call our Fax-on-Demand system at (408) 252-7215 and request document #113 to receive 4D Server information, or call ACI for your free 4D Server demo. After experiencing 4D Server, you'll know where file-sharing database technology belongs: in the circular file.

ACI US Inc. 20883 Stevens Creek Blvd., Cupertino, CA 95014
Tel. 408 252 4444. Fax 408 252 4829. AppleLink D4444

FROM THE PUBLISHER

By Neil Ticktin, Editor-in-Chief/Publisher

Several issues ago in the June, 1994 issue, we went through a quiet change here at *MacTech*. David Williams, the official Publisher since 1992, stepped aside from that title to better reflect our division of responsibilities. Since that time, I've been working under the title of Editor-in-Chief and Publisher – with most of my duties continuing to be Publisher related. This is due to our continued expansion of editorial staff at *MacTech Magazine*.

As Publisher, I will, from time to time, write here – in the Publisher's column – so that I continue to communicate directly with you, our readers. As always, you are welcome to e-mail me. As Scott attests to in his column this month, I'm an e-mail fanatic. I've even heard that our staff throws "Fingers can now rest" parties whenever I go on vacation. So write us, let us know what you think and what you want. I do respond to just about every e-mail I get – personally.

Software Frameworks Closes MacTech Helps SFA Members

As many of you may have already heard by now, Software Frameworks Association (formerly known as MADA) has closed its doors and ceased doing business. To help soften the blow to the Macintosh developer community, we've announced that we're helping SFA members during this transition period.

MacTech to *FrameWorks* subscribers ...

For all SFA members who have issues remaining in their *FrameWorks* magazine and/or disk subscription, we will be providing two issues of *MacTech* magazine and/or disk subscription for each outstanding issue of *FrameWorks*. Those SFA members who are already *MacTech* subscribers will receive an extension to their current subscription. The transformation will take place starting with the November issue of *MacTech* (possibly the October issue – the one you are reading right now). *FrameWorks* subscribers do not need to do anything, it will all happen automatically.

More OOP in *MacTech*...

We'll be adding or enhancing our coverage of OOP, in part so we can pick up where *FrameWorks* is leaving off. We're working with Mary Elaine Califf (*FrameWorks'* Editor) to run pending *FrameWorks* articles. Furthermore, we're inviting *FrameWorks* authors to continue to write – just now for *MacTech Magazine*. You should expect to see the same *FrameWorks* type articles on a regular basis in *MacTech*. That's in addition to the "regular" articles you are used to seeing in *MacTech*.

FrameWorks back issues still available...

To make certain that *FrameWorks* articles and materials are still available to the community, *MacTech* has agreed to include some or all of these articles in a future release of the *MacTech CD-ROM*. *FrameWorks* back issues of disks, magazines and CDs are available through the MacTech Mail Order Store.

The MacTech Mail Order Store is also inviting the products formerly published and distributed by SFA to be a part of the Mail Order Store. For information or availability of any of these products, contact our Mail Order Dept.

New Exclusive Products

Starting immediately, the Mail Order Store is now the exclusive distributor of OOP-related products: AdLib, a replacement for Apple's MacApp ViewEdit; Savvy, the only scripting addition to MacApp that allows for full AppleScript attachability and editability; and MAScript (which Apple will be incorporating into the next release of MacApp). These products are now available in the Mail Order Store (with full descriptions) at the back of this issue. Check them out!

We're here to help!

We at *MacTech Magazine* have always had a lot of respect for SFA – they served the needs of an important niche for Macintosh programmers. We'll be sorry to see SFA go.

SFA provided a forum for people to meet, discuss, and publish a continual stream of great ideas. Now that SFA is gone, *MacTech* would like to help the community keep these ideas flowing. What we want to know from you is: how can we help?

For example, do the SFA affiliates need support? What directions does the community want frameworks to go? In other words, what's on your mind?

New Folios!

Periodically, we get comments from readers about the formatting of the magazine. Many of you have noticed the new style we introduced with the August issue. In our quest to continually improve *MacTech*, we've made another change – this time to our folios (the thing with the page number and the name of the magazine at the bottom of the page). Starting this month, we've added the title of the article to the folio. This should make searching for an article easier.

As always, let us know what you think.

Why So Many Have Moved to MicroGuard Macintosh Software Copy Protection:

"Based on our experience, after extensive testing and evaluation, MicroGuard has been selected by **Quark** as its major supplier of Macintosh copy protection keys because the MicroGuard key and the MicroGuard organization deliver on their promises."

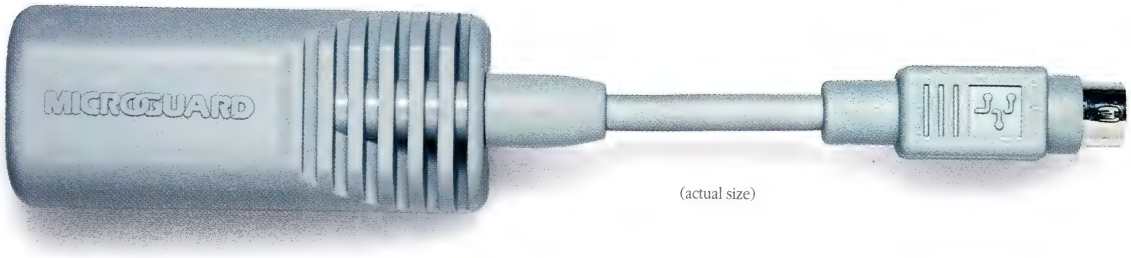
Dave Schroeder
Quark, Inc.

"Unfortunately, copy protection is a necessary evil. Fortunately, MicroGuard is exactly what we were looking for when we searched for a copy protection key for **After Effects**. It is secure, flexible, user friendly, and it is backed up by a very professional and pleasant organization that is always ready and able to provide assistance."

David Herbstman
CoSA Division
Aldus Corporation

"Because **Live Picture**, a new high-end image compositing and retouching tool, is protected with a hardware lock, we looked at all the available options. We chose MicroGuard because of its extraordinary security, compatibility, and flexibility. MicroGuard, the company, meets all our production requirements, sometimes delivering orders to us on the same day when it really counted. We enjoy doing business with MicroGuard."

Shawn Steiner
HSC Software



(actual size)

Contact us and we will be pleased to show you firsthand why MicroGuard is the leader in Macintosh copy protection worldwide

To receive more information including price lists, to order a Developer's Kit, to receive a free CD ROM about MicroGuard, or for the name of our local distributor, please contact us at:

MicroGuard USA: Tel: 303-320-1628 ■ Fax: 303-320-1599 ■ AppleLink: M.GUARD
International: Tel: 972-2-868-180 ■ Fax: 972-2-868-120 ■ AppleLink: MICROGUARD

The MICROGUARD™ Family: *People and Products*

By Dave Mark, MacTech Magazine Regular Contributing Author

Working With Color

This month's column combines two of my favorite activities: working with color and rewriting Primer, Volume II code (bringing it from the Pliocene era to full PowerPC squishiness). This month's program is a floor to ceiling rewrite of ColorTutor. ColorTutor is a hands-on color blending environment. You specify the foreground and background colors and patterns, then select a Color Quickdraw drawing mode. ColorTutor uses CopyBits() to mix the foreground and background colors. Figure 1 shows a sample.

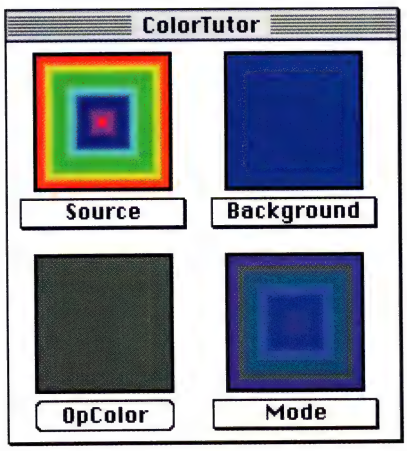


Figure 1. The ColorTutor window.

ColorTutor first copies the Background image to the lower-right rectangle, then copies the Source image on top of the Background using the current Mode and OpColor. Since this program is so large, we'll get into the details in next month's column. For now, we'll focus on putting the project together and getting ColorTutor up and running.

The ColorTutor Resources

ColorTutor uses six different resource types: an ALRT, a CNTL, a DITL, an MBAR, a MENU, and a WIND. Start by creating a folder named `ColorTutor` in your `Projects` folder. Next, fire up ResEdit or Resorcerer and create a new file named `ColorTutor.π.rsrc` in the `ColorTutor` folder.

Create an `ALRT` resource with an ID of 128, a top of 40, left of 40, bottom of 156 and right of 332. Make sure the DITL ID is set to 128.

Next, create a DITL with an ID of 128. Figure 2 shows the specifications for item 1, the OK button, and Figure 3 shows the specs for item 2, the static text field. The alert you just created is used to display an error message.

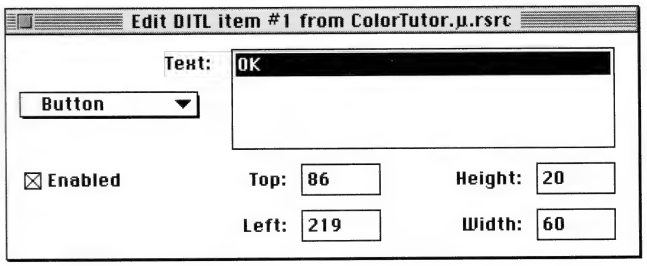


Figure 2. Specifications for the OK button.

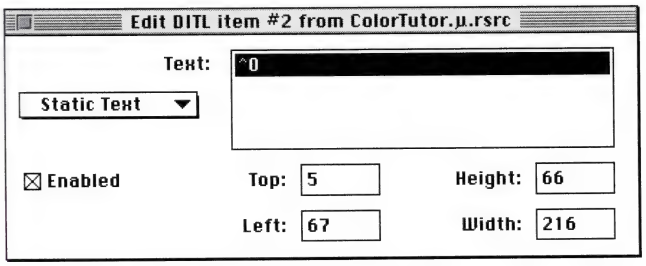


Figure 3. Specifications for item 2, the static text field.

Next, you'll create a CNTL resource with an ID of 128. The CNTL will be used to implement the **OpColor** button in the lower-left corner of the ColorTutor window. The ProcID of 0 specifies a pushButtonProc control.

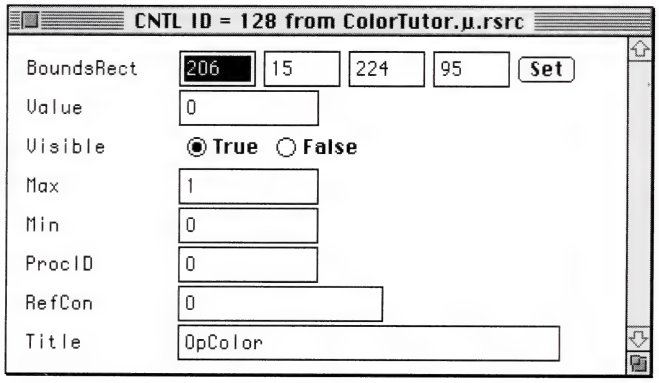


Figure 4. Specifications for the CNTL resource.

Now create an MBAR resource with an ID of 128. Add the menu IDs 128, 129, and 130 (the , **File**, and **Edit** menus) to the MBAR. Though we'll be creating 5 menus, don't be fooled. Only the first three will be added to the menu bar.

Next, you'll create five MENU resources. The first four are shown in Figure 5, and the fifth in Figure 6. MENUs 128, 129, and 130 will be used to create the menu bar. The last two implement the ColorTutor popup menus. Note that the popup menus don't have titles. Note also that MENU 132 has 17 items including the separator line (the 9th item).

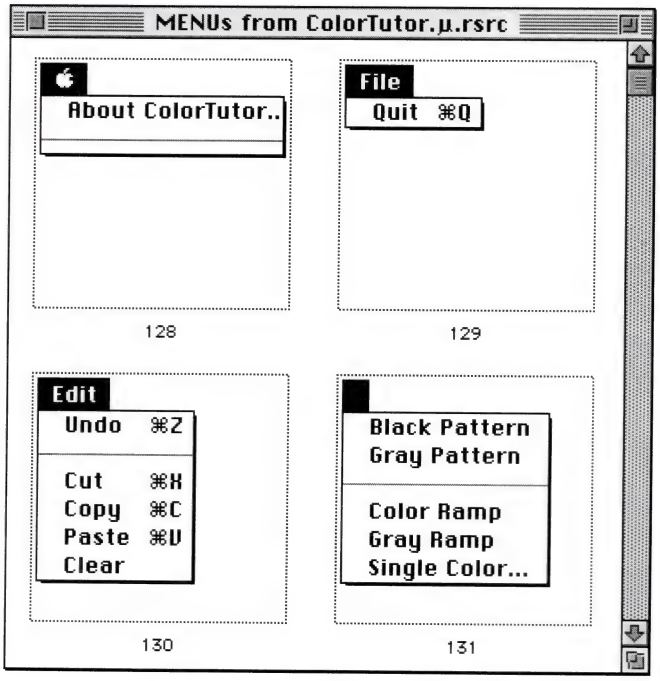


Figure 5. Specifications for the first four MENUs.

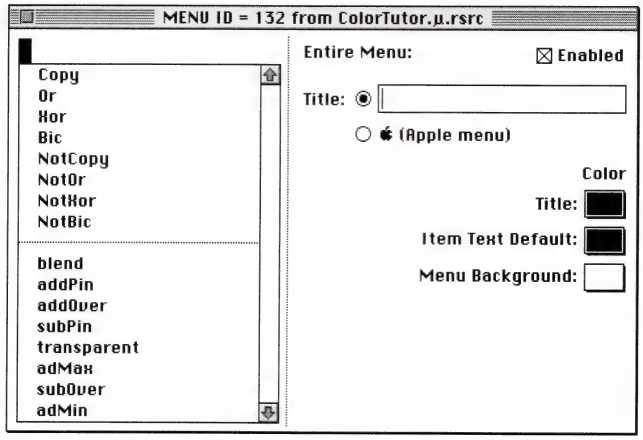


Figure 6. Specifications for the two popup MENUs.

The last resource is a WIND with a resource ID of 128. Figure 7 shows the ResEdit WIND editing screen for my WIND. This WIND implements the main ColorTutor window.

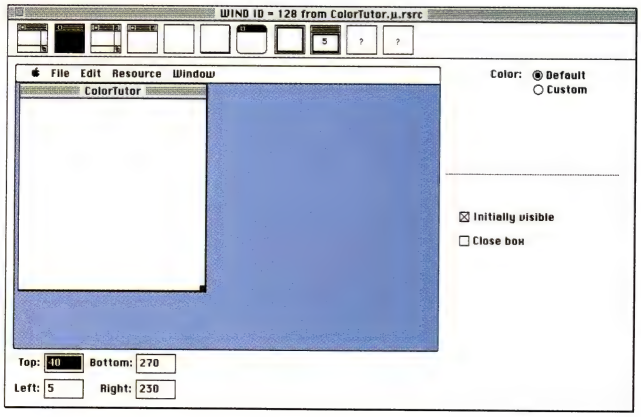


Figure 7. Specifications for the WIND resource.

Finally, save your changes and quit your resource editor.

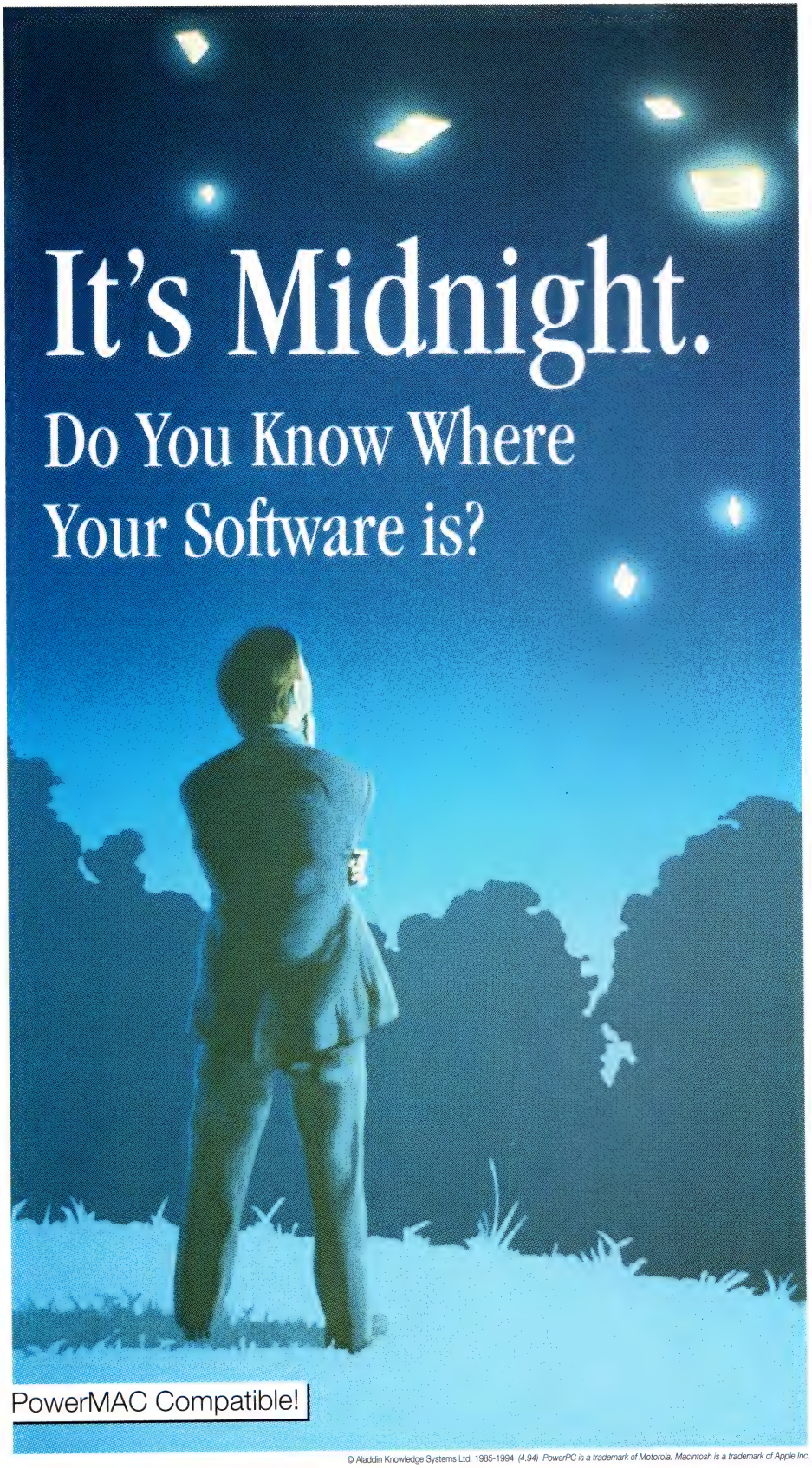
The ColorTutor Project

Next, pick your development environment and create a new project. From now on, I'll test all my source code to make sure it compiles in both THINK C and CodeWarrior, so it shouldn't matter which environment you pick. Create your new project with the name `ColorTutor.π` inside the `ColorTutor` folder.

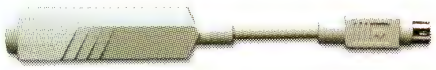
Next, add **MacTraps** to the project if you are using THINK C, or **MacOS.lib** if you are using CodeWarrior.

Finally, create a new source code file, save it as `ColorTutor.c`, and add it to the project. Here's the source code:

```
#include <Picker.h>
#include <GestaltEqu.h>


#define  kBaseResID      128
```



Bringing software into the world is a little like bringing up children. You always know where they start, but you seldom know where they'll end up. These days, with illegal use of software so common, concerned developers have good reason to worry about the products of their labor. That's where MacHASP comes in.



Like a responsible babysitter, MacHASP accompanies your software wherever it goes. With MacHASP there, your software won't run out of control. Without MacHASP, in fact, your software won't run at all.

For developers, MacHASP provides the highest level of security and reliability. For legitimate users, MacHASP is a friendly and transparent solution. Once connected, they won't even feel it's there.

And if your child wants to play with its friends, a single Net-MacHASP lets it run free around a local area network. But always under your supervision and control.

The HASP family of software protection products. Because software developers have enough sleepless nights already.

Call now for your MacHASP evaluation kit.

ALADDIN

The Professional's Choice

North America	**Aladdin Software Security Inc** The Empire State Building 350 Fifth Avenue, Suite 7204 New York, NY 10118, USA Tel: (800) 223 4277, 212-564 5678 Fax: 212-564 3377
Intl Office	**Aladdin Knowledge Systems Ltd.** 15 Beit Oved St., Tel Aviv, Israel P.O.Box 11141, Tel Aviv 61110 Tel: 972-3-537 5795, Fax: 972-3-537 5796
United Kingdom	**Aladdin Knowledge Systems UK Ltd.** Tel: 0753-622266, Fax: 0753-622262
France	**Aladdin France SA** Tel: 1 40 85 98 85, Fax: 1 41 21 90 56

member of

■ **Australia** Conlab 3 8985685 ■ **Czech** Atlas 2 766085 ■ **Chile** Micrologica 2 222 1388 ■ **Denmark** Berendsen 39 577100
■ **Egypt** Zeineldein 2 3604632 ■ **Finland** ID-Systems 0 870 3520 ■ **Germany** CSS 201 278804 ■ **Greece** Unibrain 1 6856320
■ **Italy** Partner Data 2 26147380 ■ **Japan** Athena, 3 58 213284 ■ **Korea** Dae-A 2 848 4481 ■ **New Zealand** Training , 4 5666014
■ **Poland** Systherm 61 475065 ■ **Portugal** Futurmatica 1 4116269 ■ **South Africa** D Le Roux, 11 886 4704 ■ **Spain** PC Hardware, 3 4493193
■ **Switzerland** Opag 61 7112245 ■ **Taiwan** Teco 2 555 9676 ■ **Turkey** Mikrobeta 312 467 7504

```
#define kErrorALRTid  128
#define kNullFilterProc    NULL
#define kMoveToFront  (WindowPtr)-1L
#define  kNotNormalMenu    -1
#define kSleep        60L

#define mApple        kBaseResID
#define iAbout        1

#define mFile       kBaseResID+1
#define iQuit       1

#define  mColorsPopup    kBaseResID+3
#define  iBlackPattern   1
#define  iGrayPattern    2
#define  iColorRamp      4
#define  iGrayRamp       5
#define  iSingleColor    6

#define  mModePopup      kBaseResID+4
```

Globals

```
Boolean    gDone;

Rect   gSrcRect, gBackRect, gDestRect, gSrcMenuRect,
       gBackMenuRect, gModeMenuRect, gOpColorRect;

int    gSrcPattern, gBackPattern, gCopyMode, gSrcType,
       gBackType;

RGBColor gSrcColor, gBackColor, gOpColor;

MenuHandle gSrcMenu, gBackMenu, gModeMenu;
```

Functions

```
void   ToolboxInit( void );
void   MenuBarInit( void );
void   CreateWindow( void );
void   SetUpGlobals( void );
void   EventLoop( void );
void   DoEvent( EventRecord *eventPtr );
void   HandleMouseDown( EventRecord *eventPtr );
void   HandleMenuChoice( long menuChoice );
void   HandleAppleChoice( short item );
void   HandleFileChoice( short item );
void   DoUpdate( WindowPtr window );
void   DrawContents( WindowPtr window );
void   DrawColorRamp( Rect *rPtr );
void   DrawGrayRamp( Rect *rPtr );
void   DrawLabel( Rect *boundsPtr, Str255 s );
void   DoContent( WindowPtr window, Point globalPoint );
void   UpdateSrcMenu( void );
void   UpdateBackMenu( void );
void   UpdateModeMenu( void );
void   DoSrcChoice( short item );
void   DoBackChoice( short item );
void   DoModeChoice( short item );
short    DoPopup( MenuHandle menu, Rect *boundsPtr );
Boolean    PickColor( RGBColor *colorPtr );
Boolean    HasColorQD( void );
void   DoError( Str255 errorString );
```

main

```
void main( void )
{
  ToolboxInit();
  MenuBarInit();

  if ( ! HasColorQD() )
    DoError( "\pThis machine does not support Color QuickDraw!" );

  CreateWindow();
  SetUpGlobals();

  EventLoop();
}
```

ToolboxInit

```
void ToolboxInit( void )
{
  InitGraf( &qd.thePort );
  InitFonts();
  InitWindows();
  InitMenus();
  TEInit();
  InitDialogs( 0L );
  InitCursor();
}
```

MenuBarInit

```
void MenuBarInit( void )
{
  Handle     menuBar;
  MenuHandle    menu;

  menuBar = GetNewMBar( kBaseResID );

  if ( menuBar == NULL )
    DoError( "\pCouldn't load the MBAR resource..." );

  SetMenuBar( menuBar );

  menu = GetMHandle( mApple );
  AddResMenu( menu, 'DRVR' );

  DrawMenuBar();
}
```

CreateWindow

```
void CreateWindow( void )
{
  WindowPtr  window;

  window = GetNewCWindow( kBaseResID, NULL, kMoveToFront );

  GetNewControl( kBaseResID, window );

  SetPort( window );

  TextFont( systemFont );
}
```

SetUpGlobals

```
void SetUpGlobals( void )
{
  SetRect( &gSrcRect, 15, 6, 95, 86 );
  SetRect( &gBackRect, 125, 6, 205, 86 );
  SetRect( &gDestRect, 125, 122, 205, 202 );
  SetRect( &gOpColorRect, 15, 122, 95, 202 );

  SetRect( &gSrcMenuRect, 7, 90, 103, 108 );
  SetRect( &gBackMenuRect, 117, 90, 213, 108 );
  SetRect( &gModeMenuRect, 117, 206, 213, 224 );

  gSrcPattern = iBlackPattern;
  gBackPattern = iBlackPattern;

  gCopyMode = srcCopy;

  gSrcColor.red = 65535;
  gSrcColor.green = gSrcColor.blue = 0;
  gSrcType = iSingleColor;

  gBackColor.blue = 65535;
  gBackColor.red = gBackColor.green = 0;
  gBackType = iSingleColor;

  gOpColor.green = 32767;
  gOpColor.red = 32767;
  gOpColor.blue = 32767;
  OpColor( &gOpColor );

  gSrcMenu = GetMenu( mColorsPopup );
  InsertMenu( gSrcMenu, kNotNormalMenu );

  gBackMenu = GetMenu( mColorsPopup );
  InsertMenu( gBackMenu, kNotNormalMenu );

  gModeMenu = GetMenu( mModePopup );
  InsertMenu( gModeMenu, kNotNormalMenu );
}
```

We will not be emulated.

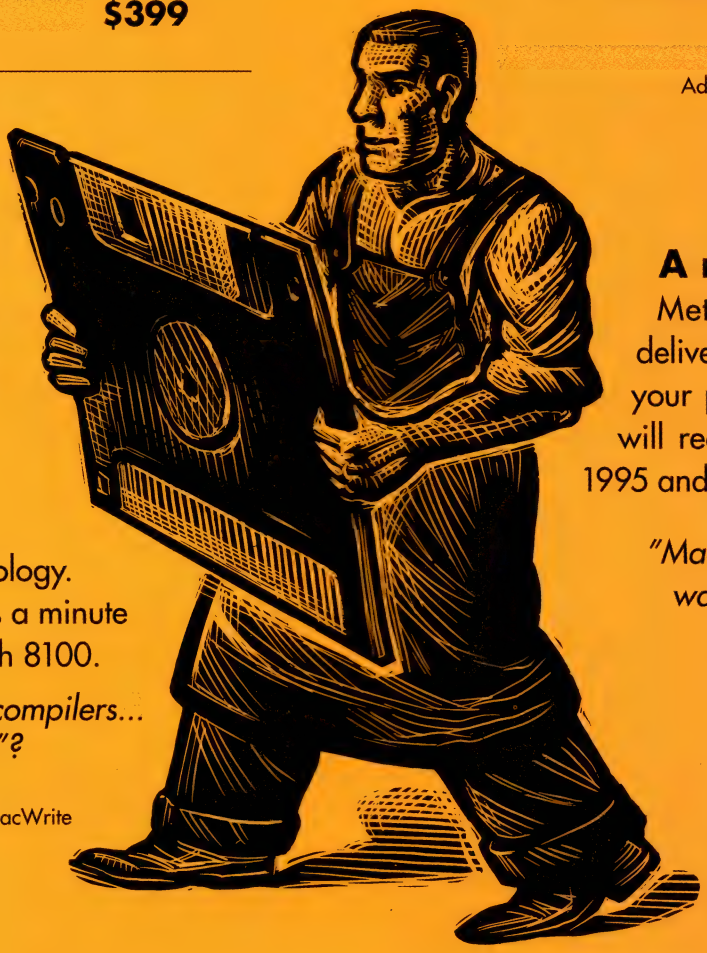
Metrowerks CodeWarrior. The only native development environment for the Power Macintosh is here.

Bronze For 68K Mac	**$ 99**
Gold For Power & 68K Mac	**$399**

Power Macintosh and 68K Macintosh application framework – Metrowerks PowerPlant. ToolServer, SourceServer and other developer utilities from Apple Development Products. It's all here.

"Without the Metrowerks PowerPC compiler it would be virtually impossible to develop Adobe Illustrator for the Macintosh on the PowerPC."

Don Melton
Software Engineering Leader
Adobe Illustrator for Power Macintosh
Adobe Inc.



Metrowerks CodeWarrior. A new way to buy.

Metrowerks CodeWarrior delivers 3 times a year. With your purchase of CW4, you will receive CW5 in January 1995 and CW6 in May 1995.

"Mac developers have been waiting a long time for an environment like Metrowerks CodeWarrior."

Stanley Crane
General Manager R&D,
cc: Mail Division
Lotus Development Corp.

Metrowerks CodeWarrior is a screamer.

Experience RISC technology. Compile 200,000 lines a minute on the Power Macintosh 8100.

"Great company, fast compilers... how can you beat that"?

Lee Richardson
Development Manager, MacWrite
Claris Corporation

Metrowerks CodeWarrior is awesome.

Native C++ and C compilers for the Power Macintosh. Native C++, C and Pascal compilers for the 68K Macintosh. 68K Macintosh-hosted PowerPC compilers. Power Macintosh and 68K Macintosh source-level debuggers. A new

"Call MacTech"
310-575-4343

metrowerks

EventLoop

```
void EventLoop( void )
{
  EventRecord  event;

  gDone = false;
  while ( gDone == false )
  {
    if ( WaitNextEvent( everyEvent, &event, kSleep, NULL ) )
      DoEvent( &event );
  }
}
```

DoEvent

```
void DoEvent( EventRecord *eventPtr )
{
  char  theChar;

  switch( eventPtr->what )
  {
    case mouseDown:
      HandleMouseDown( eventPtr );
      break;
    case keyDown:
    case autoKey:
      theChar = eventPtr->message & charCodeMask;

      if ( (eventPtr->modifiers & cmdKey) != 0 )
        HandleMenuChoice( MenuKey( theChar ) );
      break;
    case updateEvt:
      DoUpdate( (WindowPtr)eventPtr->message );
      break;
  }
}
```

HandleMouseDown

```
void HandleMouseDown( EventRecord *eventPtr )
{
  WindowPtr    window;
  short      thePart;
  long     menuChoice;

  thePart = FindWindow( eventPtr->where, &window );

  switch ( thePart )
  {
    case inMenuBar:
      menuChoice = MenuSelect( eventPtr->where );
      HandleMenuChoice( menuChoice );
      break;
    case inSysWindow :
      SystemClick( eventPtr, window );
      break;
    case inContent:
      if ( window != FrontWindow() )
        SelectWindow( window );
      else
        DoContent( window, eventPtr->where );
      break;
    case inDrag :
      DragWindow( window, eventPtr->where, &qd.screenBits.bounds );
      break;
  }
}
```

HandleMenuChoice

```
void HandleMenuChoice( long menuChoice )
{
  short  menu;
  short  item;

  if ( menuChoice != 0 )
  {
    menu = HiWord( menuChoice );
    item = LoWord( menuChoice );

    switch ( menu )
    {
      case mApple:
        HandleAppleChoice( item );
        break;
      case mFile:
        HandleFileChoice( item );
        break;
    }
    HiliteMenu( 0 );
  }
}
```

HandleAppleChoice

```
void HandleAppleChoice( short item )
{
  MenuHandle appleMenu;
  Str255   accName;
  short    accNumber;

  switch ( item )
  {
    case iAbout:
      SysBeep( 20 );
      break;
    default:
      appleMenu = GetMHandle( mApple );
      GetItem( appleMenu, item, accName );
      accNumber = OpenDeskAcc( accName );
      break;
  }
}
```

HandleFileChoice

```
void HandleFileChoice( short item )
{
  switch ( item )
  {
    case iQuit:
      gDone = true;
      break;
  }
}
```

DoUpdate

```
void DoUpdate( WindowPtr window )
{
  BeginUpdate( window );

  DrawContents( window );
  DrawControls( window );

  EndUpdate( window );
}
```

DrawContents

```
void DrawContents( WindowPtr window )
{
  RGBColor rgbBlack;
  Rect   source, dest;

  rgbBlack.red = rgbBlack.green = rgbBlack.blue = 0;

  if ( gSrcPattern == iBlackPattern )
    PenPat( &qd.black );
  else
    PenPat( &qd.gray );

  if ( gSrcType == iColorRamp )
    DrawColorRamp( &gSrcRect );
  else if ( gSrcType == iGrayRamp )
    DrawGrayRamp( &gSrcRect );
  else
  {
    RGBForeColor( &gSrcColor );
    PaintRect( &gSrcRect );
  }

  if ( gBackPattern == iBlackPattern )
    PenPat( &qd.black );
  else
    PenPat( &qd.gray );

  if ( gBackType == iColorRamp )
```

Gives you the Information to *Program your Best!*

The Debugger V2 & MacNosy

by Steve Jasik

Information

Control

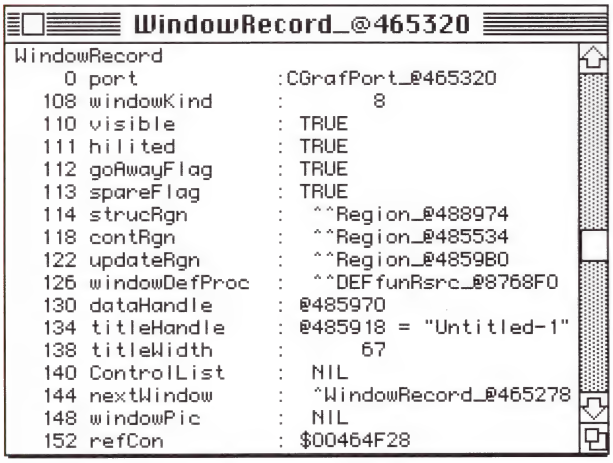
The Debugger is a low and high-level symbolic Debugger that runs in a full multi-window Macintosh environment. You can trace program execution, view the values of variables, etc. ***of both 68K and PowerPC programs***.

MacNosy is a global interactive disassembler that enables one to recover the source code of any Mac application, resource file or the ROM.

When you compare features of the different debuggers, note that *only one* has all the below features to help you get your job done, and *only one* has MacNosy to help you debug any program in a full system (6.0x or System 7.x) environment symbolically!

It is the *only* debugger to use the MMU to protect your CODE resources and the rest of the system from the program you are debugging. With MMU Protection you can find errors when they happen, not millions of instructions later! (Macintoshs with 68030 CPUs only).

The Debugger is the debugger of choice at: Adobe, Aldus, Claris, Electronic Arts, Kodak, Metrowerks, etc.



An example of a structured data display window

Its Features Include:

- **Symbolic Debugging** of *any* Macintosh program, ROM, or *code resource* (DRVRs, XCMDs, INITs, PDEFs, 4DEXs ..)
- **Source level debugging for Metrowerks & MPW** compiled programs (C++, C, Pascal, Fortran, ...), and an Incremental Build System with instant Link for superfast development.
- **Object Inspector for MacApp 3 programs**
- **Source level debugging of Think C™ projects**
- **Includes a program (CoverTest) to interactively do Code Coverage analysis for SQA testing, etc.**
- **Simultaneous Symbolic debugging of multiple "tasks"**
- **Fast Software Watchpoint** command to find clobbered variables
- **Sophisticated error check algorithms** such as Trap Discipline (Argument Checking), Handle Zapping, Heap Scramble and Heap Check to detect program errors before they become disasters
- **Structured display** of data (hypertext) with user definable structures while debugging
- **Conditional breakpoints** to help filter out redundant information
- **Continuous Animated Step Mode** to watch your program execute instruction by instruction
- **Detailed symbolic disassembly for both 680x0 and PowerPC** with symbol names, labels, cross ref maps, – make it possible to ferret out the secrets of the ROM, etc.
- **"Training Wheels"** for the PowerPC disassembler to help you learn the opcodes

The Debugger V2 & MacNosy: $350
Runs on all Macs. Call For Group prices or Updates. Visa/MC Accepted.

Available from: Jasik, APDA, Frameworks or ComputerWare (800-326-0092).

Jasik Designs • 343 Trenton Way, Menlo Park, California 94025 • **(415) 322-1386**
Internet: macnosy@netcom.com • Applelink: D1037

```
    DrawColorRamp( &gBackRect );
  else if ( gBackType == iGrayRamp )
    DrawGrayRamp( &gBackRect );
  else
  {
    RGBForeColor( &gBackColor );
    PaintRect( &gBackRect );
  }

  PenPat( &qd.black );

  RGBForeColor( &gOpColor );
  PaintRect( &gOpColorRect );

  RGBForeColor( &rgbBlack );
  DrawLabel( &gSrcMenuRect, "\pSource" );
  DrawLabel( &gBackMenuRect, "\pBackground" );
  DrawLabel( &gModeMenuRect, "\pMode" );

  PenSize( 2, 2 );
  FrameRect( &gSrcRect );
  FrameRect( &gBackRect );
  FrameRect( &gDestRect );
  FrameRect( &gOpColorRect );

  PenNormal();

  source = gBackRect;
  InsetRect( &source, 2, 2 );

  dest = gDestRect;
  InsetRect( &dest, 2, 2 );

  CopyBits( (BitMap *)&(((CGrafPtr)window)->portPixMap),
        (BitMap *)&(((CGrafPtr)window)->portPixMap),
        &source, &dest, srcCopy, NULL );

  source = gSrcRect;
  InsetRect( &source, 2, 2 );

  CopyBits( (BitMap *)&(((CGrafPtr)window)->portPixMap),
        (BitMap *)&(((CGrafPtr)window)->portPixMap),
        &source, &dest, gCopyMode, NULL );
}
```

DrawColorRamp

```
void DrawColorRamp( Rect *rPtr )
{
  long   numColors, i;
  HSVColor hsvColor;
  RGBColor rgbColor;
  Rect   r;

  r = *rPtr;

  InsetRect( &r, 2, 2 );
  numColors = ( rPtr->right - rPtr->left - 2 ) / 2;
  hsvColor.value = hsvColor.saturation = 65535;

  for ( i = 0; i < numColors; i++ )
  {
    hsvColor.hue = i * 65535 / numColors;
    HSV2RGB( &hsvColor, &rgbColor );
    RGBForeColor( &rgbColor );

    FrameRect( &r );
    InsetRect( &r, 1, 1 );
  }
}
```

DrawGrayRamp

```
void DrawGrayRamp( Rect *rPtr )
{
  long   numColors, i;
  RGBColor rgbColor;
  Rect   r;

  r = *rPtr;
  InsetRect( &r, 2, 2 );
  numColors = ( rPtr->right - rPtr->left - 2 ) / 2;

  for ( i = 0; i < numColors; i++ )
  {
    rgbColor.red = i * 65535 / numColors;
    rgbColor.green = rgbColor.red;
    rgbColor.blue = rgbColor.red;

    RGBForeColor( &rgbColor );

    FrameRect( &r );
    InsetRect( &r, 1, 1 );
  }
}
```

DrawLabel

```
void DrawLabel( Rect *boundsPtr, Str255 s )
{
  Rect r;
  int    size;

  r = *boundsPtr;
  r.bottom -= 1;
  r.right -= 1;
  FrameRect( &r );

  MoveTo( r.left + 1, r.bottom );
  LineTo( r.right, r.bottom );
  LineTo( r.right, r.top + 1 );

  size = boundsPtr->right - boundsPtr->left - StringWidth(s);
  MoveTo( boundsPtr->left + size / 2, boundsPtr->bottom - 6);

  DrawString( s );
}
```

DoContent

```
void DoContent( WindowPtr window, Point globalPoint )
{
  int        choice;
  ControlHandle control;
  RGBColor   rgbColor;
  Point      p;

  p = globalPoint;
  GlobalToLocal( &p );

  if ( FindControl( p, window, &control ) )
  {
    if ( TrackControl( control, p, NULL ) )
    {
      rgbColor = gOpColor;

      if ( PickColor( &rgbColor ) )
      {
        gOpColor = rgbColor;

        InvalRect( &gOpColorRect );
        InvalRect( &gDestRect );

        OpColor( &gOpColor );
      }
    }
  }
  else if ( PtInRect( p, &gSrcMenuRect ) )
  {
    UpdateSrcMenu();

    choice = DoPopup( gSrcMenu, &gSrcMenuRect );

    if ( choice > 0 )
    {
      DoSrcChoice( choice );

      InvalRect( &gSrcRect );
      InvalRect( &gDestRect );
    }
  }
  else if ( PtInRect( p, &gBackMenuRect ) )
  {
    UpdateBackMenu();

    choice = DoPopup( gBackMenu, &gBackMenuRect );
```

```
    if ( choice > 0 )
    {
      DoBackChoice( choice );

      InvalRect( &gBackRect );
      InvalRect( &gDestRect );
    }
  }
  else if ( PtInRect( p, &gModeMenuRect ) )
  {
    UpdateModeMenu();

    choice = DoPopup( gModeMenu, &gModeMenuRect );

    if ( choice > 0 )
    {
      DoModeChoice( choice );

      InvalRect( &gDestRect );
    }
  }
}
```

UpdateSrcMenu

```
void UpdateSrcMenu( void )
{
  int    i;

  for ( i = 1; i <= 6; i++ )
    CheckItem( gSrcMenu, i, false );

  if ( gSrcPattern == iBlackPattern )
    CheckItem( gSrcMenu, iBlackPattern, true );
  else
    CheckItem( gSrcMenu, iGrayPattern, true );

  if ( gSrcType == iColorRamp )
    CheckItem( gSrcMenu, iColorRamp, true );
  else if ( gSrcType == iGrayRamp )
    CheckItem( gSrcMenu, iGrayRamp, true );
  else if ( gSrcType == iSingleColor )
    CheckItem( gSrcMenu, iSingleColor, true );
}
```

UpdateBackMenu

```
void UpdateBackMenu( void )
{
  int    i;

  for ( i = 1; i <= 6; i++ )
    CheckItem( gBackMenu, i, false );

  if ( gBackPattern == iBlackPattern )
    CheckItem( gBackMenu, iBlackPattern, true );
  else
    CheckItem( gBackMenu, iGrayPattern, true );

  if ( gBackType == iColorRamp )
    CheckItem( gBackMenu, iColorRamp, true );
  else if ( gBackType == iGrayRamp )
    CheckItem( gBackMenu, iGrayRamp, true );
  else if ( gBackType == iSingleColor )
    CheckItem( gBackMenu, iSingleColor, true );
}
```

UpdateModeMenu

```
void UpdateModeMenu( void )
{
  int    i;

  for ( i = 1; i <= 17; i++ )
    CheckItem( gModeMenu, i, false );

  if ( ( gCopyMode >= 0 ) && ( gCopyMode <= 7 ) )
    CheckItem( gModeMenu, gCopyMode + 1, true );
  else
    CheckItem( gModeMenu, gCopyMode - 22, true );
}
```

DoSrcChoice

```
void DoSrcChoice( short item )
{
  RGBColor rgbColor;

  switch ( item )
  {
    case iBlackPattern:
    case iGrayPattern:
      gSrcPattern = item;
      break;
    case iColorRamp:
    case iGrayRamp:
      gSrcType = item;
      break;
    case iSingleColor:
      gSrcType = iSingleColor;
      rgbColor = gSrcColor;

      if ( PickColor( &rgbColor ) )
        gSrcColor = rgbColor;
      break;
  }
}
```

DoBackChoice

```
void DoBackChoice( short item )
{
  RGBColor rgbColor;

  switch ( item )
  {
    case iBlackPattern:
    case iGrayPattern:
      gBackPattern = item;
      break;
    case iColorRamp:
    case iGrayRamp:
      gBackType = item;
      break;
    case iSingleColor:
      gBackType = iSingleColor;
      rgbColor = gBackColor;

      if ( PickColor( &rgbColor ) )
        gBackColor = rgbColor;
      break;
  }
}
```

DoModeChoice

```
void DoModeChoice( short item )
{
  if ( ( item >= 1 ) && ( item <= 8 ) )
    gCopyMode = item - 1;
  else
    gCopyMode = item + 22;
}
```

DoPopup

```
short  DoPopup( MenuHandle menu, Rect *boundsPtr )
{
  Point  corner;
  long theChoice = 0L;

  corner.h = boundsPtr->left;
  corner.v = boundsPtr->bottom;

  LocalToGlobal( &corner );

  InvertRect( boundsPtr );

  theChoice = PopUpMenuSelect(menu,corner.v-1,corner.h+1,0);
  InvertRect( boundsPtr );
  return( LoWord( theChoice ) );
}
```

PickColor

```
Boolean  PickColor( RGBColor *colorPtr )
{
  Point  where;
```

The professional's database engine

CXbase Pro

for

Macintosh and Power Macintosh

Based on C-Index Pro, the database engine used by **Apple AOS** for their **eWorld** content publishing software, *CXbase Pro* has all the features and power of C-Index Pro, plus additional new capabilities. *CXbase Pro* includes a flexible select module, access to both logical and absolute record numbers, and more.

CXbase Pro is designed as a "pure" database engine. The philosophy behind *CXbase Pro* is to provide a simple, consistent, and powerful API that lets you work in whatever way you choose. *CXbase Pro* is delivered with full source code, giving you maximum flexibility and security. The *CXbase Pro* library provides both multi-keyed data record handling, and flexible BLOB storage where you define the data format. And *CXbase Pro* has been engineered from the ground up for maximum performance.

What you get:

- **Unrivaled speed**
  CXbase Pro has been fine-tuned to be the fastest database engine available, regardless of the size of the data. Store large multi-media data without performance penalties.
- **Royalty-free license agreement**
  No royalties on applications developed with *CXbase Pro*. Period.
- **Complete ANSI C source code**
  With *CXbase Pro* you get the whole package, not just an object library. You have the freedom to recompile whenever you want.
- **100% cross-platform capability**
  Both the source code and the data files are completely portable. Additional versions are available for Windows, DOS, and others.

New features:

- **Powerful and flexible select module**
  CXbase Pro lets you create multiple kinds of selections, and provides a callback mechanism that allows you to both display the progress, and capture the data as it is selected. And as with all other aspects of *CXbase Pro*, you have full control over the operation of the select routine.
- **Access to logical and absolute record numbers**
  You can access data records not only by key, but also by logical and absolute record numbers. This gives you maximum flexibility as to how you retrieve and display your data.

For the Power Macintosh:

- **Pure native operation**
  CXbase Pro runs completely native on the Power Macintosh. Performance is up to 400% of the fastest Quadra!
- **Mixed-mode support**
  Full support is provided for calling the native *CXbase Pro* library from a 68K application. Now you can immediately gain the speed benefits of the Power Macintosh without having to port all your code at once. **MacApp 3.0.1 users take note!**

$49 Full-featured demo available

CXbase Pro also comes in a demo version. You get the complete manual, and a full-featured library limited only by file size.You can try all the functions, and use it for as long as you like.

For full pricing, how to order, and more information contact:

TSE International **AppleLink: TSE.INT**
Taandwarsstraat 51
1013 BV Amsterdam tel: +31 20 638-6507
Holland fax: +31 20 620-4933

FREE INSTALLER!

To: **Mac Developers & Product Managers**

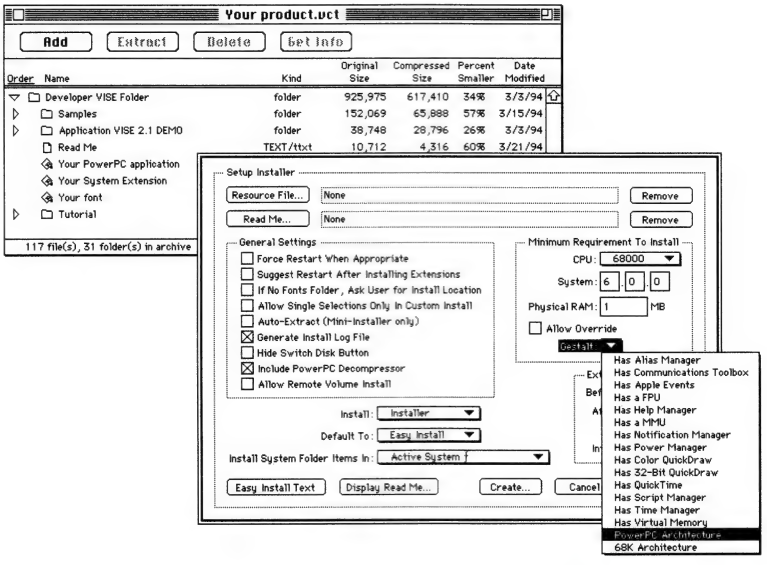
From: **MindVision (Creators of Stacker for Macintosh)**

Subject: **Free Copy of DEVELOPER VISE 3.0**

Message: **We've got a great new installer for you.**
Here's your chance to try it for free.
No risk, no obligation, no hassle.
Call today, don't delay!

Full PowerPC Support

Integrated compression



No programming required

Fully graphical interface

Call (402) 477-3269

IF YOU PREFER, USE E-MAIL. APPLELINK, AOL: MINDVISION • COMPUSERVE 70253,1437

Join the growing list of companies who use our VISE technology: Adobe, Apple, CE Software, Claris, CompuServe, DeltaPoint, MacroMedia, Radius, Stac Electronics, Symantec, WordPerfect, and many more.

MindVision Software

MindVision Software 840 South 30th St., Suite C, Lincoln, Nebraska 68510
Voice: (402) 477-3269 Fax: (402) 477-1395 AppleLink, AOL: MindVision

VISE COMPRESSION TECHNOLOGY

```
  where.h = -1;
  where.v = -1;

  return( GetColor( where, "\pChoose a color...", colorPtr,
        colorPtr ) );
}
```

HasColorQD

```
Boolean  HasColorQD( void )
{
  unsigned char version[ 4 ];
  OSErr      err;

  err = Gestalt( gestaltQuickdrawVersion, (long *)version );

  if ( version[ 2 ] > 0 )
    return( true );
  else
    return( false );
}
```

DoError

```
void DoError( Str255 errorString )
{
  ParamText( errorString, "\p", "\p", "\p" );
  StopAlert( kErrorALRTid, kNullFilterProc );
  ExitToShell();
}
```

RUNNING COLORTUTOR

Save your code, and run ColorTutor. The ColorTutor window will appear, as shown in Figure 8.

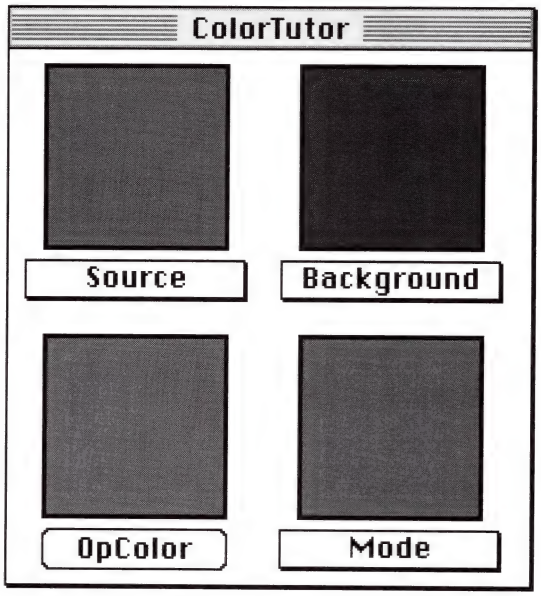


Figure 8. The ColorTutor WIndow.

The **Source** and **Background** menus are identical, as shown in Figure 9. Play with these selections till you get the source and background that you want.

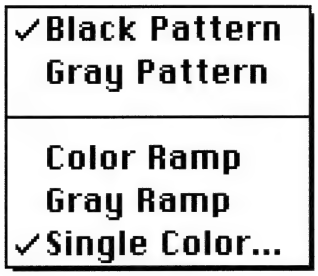


Figure 9. The Source and Background menus.

The real fun comes when you play with the **Mode** popup (Figure 10). Basically, the mode is passed as the fifth parameter to the `CopyBits()` call that copies the source rectangles over the destination rectangle which had been previously copied to the lower right corner of the ColorTutor window. Some of the modes take an OpColor, which you can set using the **OpColor** button.

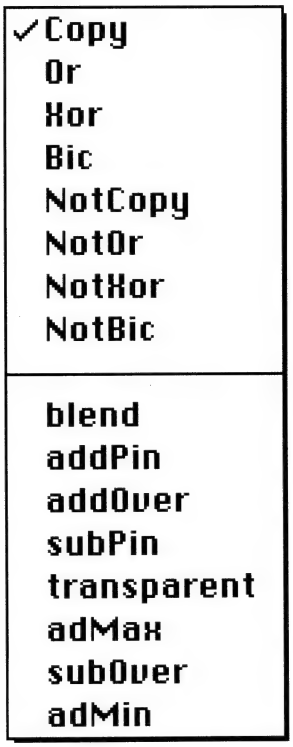


Figure 10. The Mode popup menu.

Till Next Month

Confused? Experiment! We'll get into all the hows and whys next month. Till then, read up on the Color Quickdraw transfer modes in THINK Reference and Inside Macintosh.

By Robert Munafo, Malden, MA

RGBtoYUV Using Parallel Addition

Some unique approaches to optimization

Doing more in fewer cycles...

This article contains the actual winning code for July's Color Space Conversion Programmer's Challenge. Robert had sent in his code before the deadline but for some reason the SANE calls he made during his RGBtoYUVInit routine caused both my Macs to crash. I wasn't able to identify the exact cause of the crash other than to witness that it wasn't his code. I suspect it had to do with Omega SANE's backpatching (self-modifying code) but I'm not sure. In any case, Robert was given a chance to submit a new version of RGBtoYUVInit only (which was not part of the timings) that didn't use SANE. He did so and ended up being about 27% faster than the published winner Bob Noll. As you will see, Robert's explanation and use of parallel addition is excellent (and fast!). I highly recommend studying it if you need to do fast matrix multiplication with constant coefficients.

– Mike Scanlin

When I created my entry for the July 1994 Programmers' Challenge, I used some novel optimization techniques which are explained here. For background material, see the contest statement in the July 1994 issue, page 44, and the results presented in the September 1994 issue.

I will briefly restate the challenge. It involved converting a large number of [R,G,B] values into [Y,U,V] (a color system used in JPEG and NTSC, among others) using the formula:

```
Y     0.29900000   0.58700000   0.11400000     R
U  = -0.16873590  -0.33126410   0.50000000  *  G
V     0.50000000  -0.41868760  -0.08131241     B
```

Each entry consisted of an init routine that would not be timed and an RGBtoYUV routine that would take arrays of [R,G,B] values and output arrays of [Y,U,V] values. As always in the Programmers' Challenge, accuracy is most important, followed by speed, code size, and elegance.

Analysis of Rounding

The first problem to solve involved figuring out how various types of rounding would affect computed results. The challenge required producing output that was equivalent to the results that would be produced when infinite precision is used. Fractions of N.5 (e.g. 2.5 or -2.5) would be rounded down, and anything else would be rounded to nearest. The problem clearly required the use of limited-precision integer math, so I had to figure out how much precision was necessary to produce acceptable results.

I conducted a brute-force search and determined that out of all possible Y, U, and V values produced by the transform matrix, the closest fraction to N.5 was N.499000 or N.501000. In

Robert Munafo – Robert works for VideoGuide, a startup in the Boston area. Prior to that, he developed drivers and embedded software for GCC Technologies' printer products. Robert's been writing free Mac software since 1984. One of the first public-domain games for the Mac, *Missile,* continues to run on every new model Apple releases! He also became well-known for his shareware effort *Orion* and the free utility *Icon Colorizer.* He spends most of his spare time on the Mac gronking the inner loops of the Mandelbrot Set and various compute-intensive simulations. He awaits the day when massively parallel *desktop* computers will surpass the TFLOPS (trillion floating-point operations per second) milestone. You can reach him via e-mail at mrob@world.std.com.

other words, to distinguish an N.5 result from all other results, the math must be precise enough to distinguish differences as small as 0.001, or 2^{-10}. Since 8 bits are used for the integer position of the answer, at least 18 bits are needed. The simplest way to get the right answer is to use 19 bits (or more) to compute the fraction, then add 0.4995, then chop off the fraction. This works because 0.4995 equals 0.5 - (0.001 / 2). Here are three examples:

```
2.501 + 0.4995 = 3.0005   ->   3
2.500 + 0.4995 = 2.9995   ->   2
2.499 + 0.4995 = 2.9985   ->   2
```

When considering negative values, there were technically two ways to interpret the problem's statement of ".5 rounding down to zero". The more likely interpretation is that we should round towards zero, with -2.5 rounding up to -2.0 and 2.5 rounding down to 2.0. However, consider what happens to the U values as the following sequence of [R,G,B] triples is transformed:

```
[R,G,B]    U    Rounded
 8,8,2   -3.0     -3
 8,8,3   -2.5     -2
 8,8,4   -2.0     -2
 8,8,5   -1.5     -1
 8,8,6   -1.0     -1
 8,8,7   -0.5      0
 8,8,8   +0.0      0
 8,8,9    0.5      0
 8,8,10   1.0      1
 8,8,11   1.5      1
 8,8,12   2.0      2
```

As you can see, what ought to be a steady sequence of U values {-3, -2, -2, -1, -1, 0, 0, 1, 1, 2, ...} gets an added 0. This would be noticeable in certain "fountains" or smooth gradations of color – a small banding artifact would appear whenever the U or V axis is crossed in areas where R is equal to G. (This would be more noticeable after repeated transformations back and forth between PICT and JPEG – an average of 0.5 units of negative U would be lost each time). For this reason I decided to round towards negative infinity.

Algorithms & Techniques

The first two are pretty obvious: integer math and array lookups. If we deal with 2's-complement values, the rounding towards infinity and 0.4995 rounding adjustment are easy to implement with integer math representing fixed-point fractions. Array lookups replace multiplication – since the generated code is for the 68000, there's no hope of doing any type of multiplication faster than array lookups.

Array lookup is simply the technique of storing a

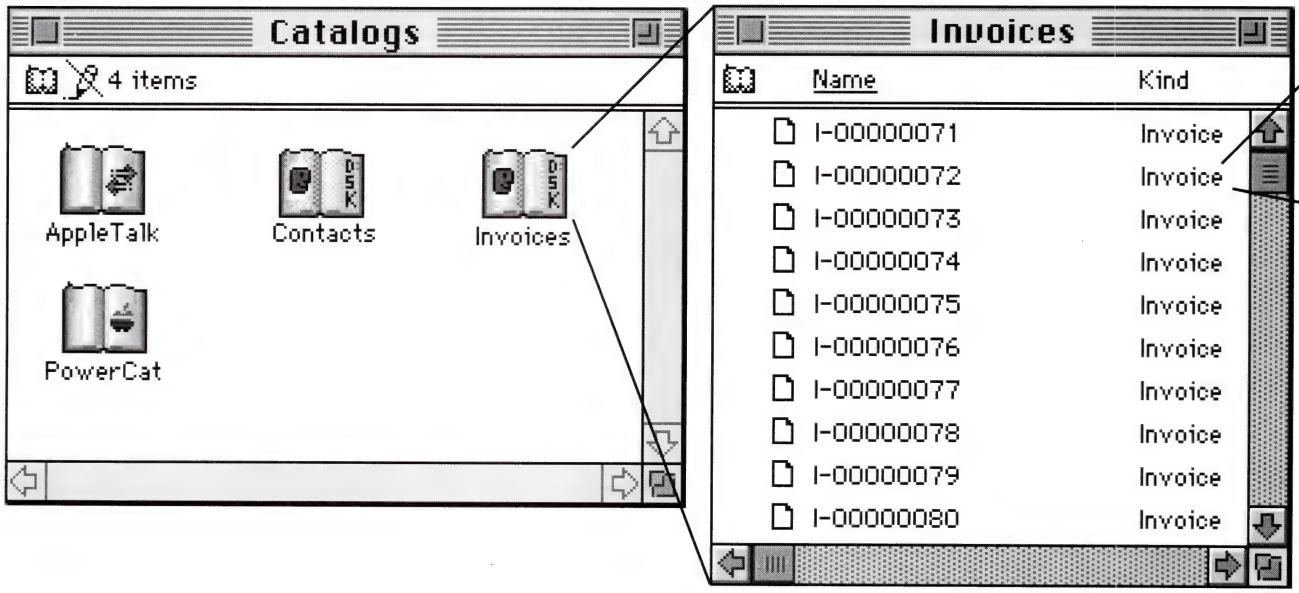
Databases in the Finder?

Unfortunately, most users must interact with relational databases at the SQL level or behind a front-end that can take months to develop.

select invoice_num,invoice_date,cust_id,cust_name from invoices,customers where invoice_id = 112;printall;

Why not view and edit your data at the Finder level? Announcing **Cataloger**™.

Cataloger brings a whole new way of working with databases to the Macintosh. The Finder™ has always provided a view of our desktop files, why not database records too?



Apple's **O**pen **C**ollaborative **E**nvironment places a catalog icon at the Finder. Open the icon up and you'll see a window like the one above. The AppleTalk catalog shows nodes on the network. The PowerCat catalog shows users and groups in a PowerShare Server.

The Contact and Invoices catalogs (for example) can show information from **4D Server**, **DAL/DAM** , **Oracle** and **Sybase** databases! If your needs are more customized, you can write AppleScript handlers to provide the contents of your catalog.

The Catalog Manager (an AOCE API) allows for **C**atalog **S**ervice **A**ccess **M**odules (CSAMs) to be created that interact with external sources.

The Finder and other applications make requests through the Catalog Manager and Cataloger does the rest. There are no limits to the amount of data brought back.

Templates provide the ability to view and edit individual records. AOCE Templates are designed with **Template Constructor™.** This powerful application let's you easily design forms to view data coming from several sources, control behavior with C and AppleScript, and view row/column data with **TableIt!™** - a powerful data display tool.

I-00000072

Invoice #: I-00000072 **Invoice Date:** 9/1/94

Company: Tall Timbers Marina **Terms:** Net 30 Days

Qty	Part #	Description	Cost Each	Line Total
1	2143	Propeller	250	250
2	3853	Drain Plugs	12.90	25.80
1	4905	Main Light assembly	150	150
8	9384	Spark Plugs	4.00	32.00
1	9999	Labor	600	600

Notes: Prepared boat for summer season. Dewinterized, tuned up and replaced main propeller. Light assembly replaced after testing front lights failed.

Subtotal: 1057.80
Tax: 52.89
Total: 1110.69

Access information and catalog definitions are stored in and protected by the AOCE **Keychain**. Your users can have one username and password for a variety of data sources!

Need more horsepower? **Database Scripting Kit™** provides AppleScript access to all of the connection information and data sources. Write AppleScript routines that send queries and parse results from within your own application.

Cataloger/Template Constructor includes:

Database Scripting Kit™ with connectors for 4D Server, DAL/DAM, Oracle, Sybase and others.

DSK Cataloger™ CSAM

Template Constructor™ with TableIt!™

Several example catalogs/templates and over 100 example AppleScripts.

Complete scriptability, native for Power Macintosh. Drag Manager and Thread Manager support.

How to contact us:

Graphical Business Interfaces, Inc.

Voice: 800-424-7714 or 219-253-8623

Fax: 219-253-7158

E-Mail: steve@gbi.com

GBI. Bringing your data into view.™

multiplication table in memory. For example, one of the values we have to multiply by in this problem is 0.587. This value is multiplied by a G (green) pixel value that is between 0 and 255. So we create an array with 256 elements; to multiply 0.587 by the value 42, we look at the 42nd entry in the array. This type of operation can be much faster than a "real" multiplication.

Parallel Addition

This is the most important optimization idea I used. Here is an example of the technique: Imagine that we're trying to add three pairs of decimal numbers:

```
  42     38     17
+ 84   + 20   + 91
----   ----   ----
  ?      ?      ?
```

and suppose that we want to accomplish this with one addition operation. We can do it by forming each row of three numbers into a single large number and adding the large numbers together:

```
  4200380017
+ 8400200091
------------
 12600580108
```

In theory, 3 10-bit binary numbers could be added in parallel this way, using 32-bit variables to hold the values. With such an approach, the RGB-to-YUV conversion would be done like this:

```
r -> lookup table ->    yr.ur.vr
g -> lookup table ->    yg.ug.vg
b -> lookup table -> + yb.ub.vb
                     ----------
                       Y  U  V
```

Each component of the RGB goes into a lookup table, to get the Y, U, and V components for that component of RGB. Then the nine Y, U, and V components are added together in one step to produce the final YUV.

Well, that's great *but*... We can only fit 10 bits of each into the 32-bit values that we're adding together, and as described above we need 19 bits to calculate each component of YUV.

One solution to this is to have two sets of YUV components for each RGB component, with the first set giving the high 10 bits of the YUV components and the second set giving the low 10 bits. However, as we'll see, we can't get this many bits and get an accurate answer.

The other solution would be to perform 64-bit math. We will discuss both options after first discussing the issue of carry bits.

Carry Bits

Going back to the decimal example above, suppose we had tried to add our three numbers together this way:

```
  42     38     17     ---->     423817
+ 84   + 20   + 91     ---->   + 842091
----   ----   ----              --------
  ?      ?      ?               1265908
```

The sum 108 from the 17+91 part of the problem has interfered with the sum 58 from the 30+28 part. The same thing happens in binary, so when we do our additions of Y,U,V we have to leave enough room for carry bits.

Let's return to the YUV problem and use 64-bit math. We have three arrays, each with 256 elements. The first array-lookup takes the R value as its index and yields a 64-bit wide value that has three 20-bit fields imbedded in it (corresponding to 0.299*R, -0.168*R, and 0.500*R) which are the "R component" of Y, U, and V; I called these fields yr, ur, and vr:

```
r -> lookup table ->    yr.ur.vr
```

The other two array-lookups are the analogous operation for G and B:

```
g -> lookup table ->    yg.ug.vg
b -> lookup table ->    yb.ub.vb
```

Then you add it together to get Y, U and V. There are three rows of figures, or in other words three figures in each column. The worst case (from a carry or overflow point of view) would occur when all three figures in a given column had the maximum possible value (which would be 2^{20-1}). This doesn't happen, but we get fairly close in the U column when R and G are both 1 and B is 255. In this case, the values ur and ug are close to 2^{20-1} and ub is 2^{19}. When you add those together you

get a value that is a little higher than 2^{21} and therefore takes 22 bits to represent. (By the way – Since we're using signed 2's-complement representations for ur, ug and ub, we can safely treat the two high bits as overflow bits and discard them.)

If we could somehow reduce it to two figures in each column, we might be able to save one carry bit. Fortunately, we can. Notice that we are allowed enough memory in our temp buffer to store 65536-element arrays. We can have one array that contains YUV components for both R and G at the same time, pre-added. We still need a separate table for the B's:

```
index_rg=((r<<8)+g)

index_rg -> table ->   yrg.urg.vrg
         b -> table ->    yb. ub. vb
```

Now we only have two numbers in each column being added. The worst case is in the V column when R is 0, G and B are 1; vrg and vb are both close to 2^{20-1} and their sum is close to 2^{21-1}, which can be represented in 21 bits. (Again, the extra bit is just a wrap-around overflow and can be safely ignored.)

Adding in Two 32-bit Parts

Now let's consider the problem of doing the YUV conversion in two 32-bit pieces, with *three* figures per column as in the original scheme. The least significant half of the computation has to generate carry bits for each of the three columns, and you need 2 carry bits for each column. As a result the best you can do is 2+8, 2+8, 2+8 with 2 unused bits; in the top half you can discard carry bits so you can manage 9, 2+9, 2+9 with 1 unused bit. If you judiciously select the placement of unused bits, everything lines up right and you get a net result of 17 "useful" bits for the computation.

Seventeen bits isn't enough, so it is now clear that we need to use the 65536-element arrays so we can get just two figures in each column to add. Here is the schematic for that:

```
                                    high          low
index_rg -> table ->             yrg.urg.vrg   yrg.urg.vrg
         b -> table -> +          yb. ub. vb    yb. ub. vb
                                  ----------    ----------
                                  Y    U    V <-(carry bits)
```

The "low" side on the right generates carry bits that are added to the "high" side to generate the result.

Consider the Y column for a moment: The actual values we need to add are 19 bits wide, and we split them into a 10-bit part and a 9-bit part – 10 bits in the high 32-bit addition, and 9 bits in the low 32-bit addition.

(By the way, notice that this is not the division of ordinal/fraction. There are *eight* bits in the ordinal (integer) part and 11 bits in the fractional part, because the entire 19-bit value needs to represent values from 0.000 to 255.999. So

those high 10 bits contain 8 ordinal bits and 2 fractional bits.)

The U and V columns are similar. Here, the values being represented are signed, with one sign bit, seven bits in the ordinal part and 11 fractional bits (12 in the case of V), to represent values from -128.000 to 127.999. Again, these bits are divided between a 10-bit most-significant part in the high 32-bit side and a 9-bit (10-bit in the case of V) least-significant part in the low 32-bit side.

64-bit Math Wins

Now we've worked out two ways to perform the RGB-to-YUV conversion, but it isn't too clear which is better. I wrote both and benchmarked them against each other, but we don't have to do that to see which one wins.

The primary disadvantage of the first approach is propagating the carries from the low 32-bit part to the high 32-bit part. It requires a shift, mask and add. The following illustrates the idea with hypothetical 21-bit values (actual values would be 32 bits wide):

```
Sum of low parts of Y, U, and V:   CyyyyyyCuuuuuuCvvvvvv
      (C represents a carry bit)
          AND it with a constant:  10000001000000100000
                          to get:  C000000C000000C000000
        now shift right by 6:      000000C000000C000000C
    and add to the "high" part:    XyyyyyyXuuuuuuXvvvvvv
(X represents an overflow bit that can be ignored)
```

With the 64-bit approach, the "low" parts in the computation consist of the entire 20-bit Y component (which needs no overflow bit because the Y computation is unsigned), plus 12 bits of the U component:

```
uuuuuuuuuuuuyyyyyyyyyyyyyyyyyyyy
```

and the "high" parts contain the remaining 8 bits of U, an overflow bit for U, all 20 bits of V, an overflow bit for V and two unused bits:

```
00XvvvvvvvvvvvvvvvvvvvvvXuuuuuuuu
```

Only one carry bit has to be handled, to handle a carry from the low 12 bits of U to the high 8 bits of U. As it turns out, we can do this without even using up one of the 32 bits!

Here's how that is accomplished: the two low parts (from the array-lookup) are added to generate a sum; if the sum is *less* than either of the addends then a carry has occurred and 1 is added to the high part. Doing this overflow test requires a compare and branch (for the test) plus an "increment" (a 68000 `ADDQ` instruction, which is faster than a normal add).

Notice that we were able to arrange the components Y, U and V in a more convenient manner. The Y component doesn't need overflow bits because each of the Y values being added together are unsigned, and we know the maximum possible value of the sum is 255.999... So we put Y on the very right, and pack 12 bits of U into the rest of the bottom part. The other 8 bits of U are the ordinal (integer) part, and we put these in the low 8 bits of the high 32-bit word. These 8 bits are what we'll write out to the U output buffer, and having them in the low 8 bits of our 32-bit word means that we won't have to do a shift to get these bits. (We still need to do a shift for the Y and V values). It might help to show the 32-bit parts again, this time with the ordinal (integer) parts of Y, U and V in uppercase letters:

```
lower part:   uuuuuuuuuuuuYYYYYYYYyyyyyyyyyyyy
upper part:   00XVVVVVVVVVvvvvvvvvvvvvXUUUUUUUU
```

It should also be pointed out that with the 64-bit approach we can compute each component with 20 bits of accuracy, one bit more than we need. More accuracy never hurts, particularly if the transform matrix might need to change. (Remember, the error analysis was valid only for the specific matrix shown at the beginning of the article.)

Instruction Scheduling and Other Optimizations

Listing 1 shows the main loop from my `RGBtoYUV` routine before I optimized it. After implementing each of the two parallel addition methods described above (and another for testing accuracy) I began optimizing the code.

I optimized quite a bit by using forced type coercion all over the place in the array index computations. This eliminated `EXT.L` instructions.

I also optimized by treating the data structure as a huge amorphous block of bytes and explicitly computing the offsets into it. This was an optimization mainly because the "offset" operation only needs to be computed twice rather than 4 times. This is possible because the first two arrays (with the 65536 elements) are both indexed by the value `((red << 8) + green)`, which means that the numbers you fetch will always be 262144 bytes apart. The same type of relation holds for the two 256-element arrays.

The most significant optimizations are what make `RGBtoYUV` so hard to read. Most of the operations have been broken up and interleaved, to minimize *pipeline stalls*. Pipeline stalls generally occur because the result of one operation is used in the next operation. By reordering operations, the code can be made to run faster without actually decreasing the number of operations.

For best results, you should begin by breaking up statements into as many small steps as possible. For example, after doing the offset-indexing changes described above my code contained the statement:

```
p = p + 262144L - index + i2;
```

Breaking this up, we get three statements:

```
p += 262144L;
p -= index;
p += i2;
```

Now you can see that each of these statements depends on the previous value of p, and generate a new value of p. Usually in this situation you would want to interleave other unrelated operations so that the value of p isn't being reused right away each time:

```
p += 262144L;
(some other operation)
p -= index;
(some other operation)
p += i2;
```

I didn't have enough unrelated operations to do this. However, I noticed that it was okay to change the value of the variable `index` (since it isn't used again) and that meant that I could think of it as

```
p = p + 262144L - (index - i2);
```

and transform *that* into:

```
p += 262144L;
index -= i2;
p -= index;
```

This solved the problem quite nicely. There is still the problem that `index` is getting used right away, but I was able to find other operations to interleave and avoid that pipeline stall as well.

In one place I was able to optimize by breaking up `x>>=12L` into two copies of `x>>=6L` . This is because shifts by more than 8 require a temporary register to be loaded with the shift amount. Normally this type of transformation wouldn't speed things up, but in this case I was able to move one of the resulting statements to reduce a pipeline stall.

Optimizations I Didn't Do (and why)

One optimization I skipped involves cache misses. When running on any machine with a data cache larger than about 1K, the performance of this algorithm will depend greatly on the gamut of source pixel values. In other words, if the source pixels are scattered all around the RGB color cube, the loads (array reads) will cause a high incidence of cache misses, with corresponding degradation in performance. On sufficiently pipelined CPUs (the '040 and PowerPC) with a large SRAM cache card this means that an algorithm with 256-entry lookup tables would outperform an algorithm with 65536-entry lookup tables.

The ideal way to address this would be with adaptive dispatching to multiple alternate algorithms. Under such a scheme, the code would process the image data in chunks, benchmarking itself with each chunk and deciding based on the performance when to switch back and forth between the 256-entry algorithm and the 65536-entry algorithm.

Unfortunately, TickCount was too coarse for this application, and I didn't want to bother with the microsecond timer.

Another optimization I skipped would handle identical source and destination buffers. It is conceivable that the caller might use the same buffers for the YUV output as for the RGB input. If a test for this were made, then three pointers could be used instead of six, allowing optimization. However, it isn't clear which of [Y,U,V] would be the same as [R], and so on; since there are six possibilities I decided it wasn't worth bothering.

If you are using an `RGBtoYUV` routine in your own program, you can probably put this optimization in quite easily.

I also refrained from unrolling the loop. After optimizing two versions of `RGBtoYUV` with the above techniques I tried loop unrolling. It improved version 1, but actually made version 2 worse. The unrolled version 1 was still slower than the non-unrolled version 2. Since unrolling might also have made it slower on the 68020 and 68030 (which have very small instruction caches) I decided to skip the idea entirely.

Benchmark Gotchas

I encountered a lot of variations in benchmark results because of cache TLB entries. For example, if you allocate 6 consecutive 1024-byte buffers for R, G, B, Y, U and V and call `RGBtoYUV` repeatedly on the contents of those buffers (without doing anything else between calls) it will usually run somewhat slower than if the buffers are further apart or are scattered around randomly in memory. There are also many dependencies related to the locations of the array indices accessed, which depend on the actual color entries used. Worst-case benchmarks usually result from filling the RGB buffers with patterns of consecutive values (as I needed to do for the code that verified correct translation of all possible [R,G,B] triples). Real RGB images would produce average results.

Also, as always I had to avoid moving the mouse to get consistent results every time. There was still a bit of fluctuation due to Ethernet traffic.

In Conclusion

By using a number of unique ideas we have arrived at an extremely fast and portable implementation of the color-space conversion, a reasonably typical iterative task. I think you will find these ideas useful in other problems – anything compute-intensive that uses integer operations. From Photoshop plug-ins and QuickTime codecs to cellular automata simulations or screen savers – the applications are virtually limitless!

I have verified the code in this article by copying the code out of this MS Word document and pasting into Think C, then running my test shell to verify proper operation. (Since I have written so many different versions of this program and had to do a lot of cutting and pasting to generate the listings given here, I was a bit uncertain if it would run.)

Listing 1

RGBtoYUV

This is the RGBtoYUV routine before all the strange optimization was applied. The final version is in the following listing.

```
void RGBtoYUV(unsigned char *ra, unsigned char *ga,
    unsigned char *ba, unsigned char *ya,
    signed char *ua, signed char *va,
    unsigned long numpix, void *pd)
{
    register signed32       index;
    register signed32       i2;
             unsigned32     i;
    register unsigned32     hi, lo_rg, lo;
    register unsigned char  *yar = ya;
    register signed char    *var = va;
    register rgb_yuv_data   *p = pd;

    for(i=0; i<numpix; i++) {
        /* Compute indexes */
        index = ((*ra++)<<8)+(*ga++);
```

```
        i2 = *ba++;

        /* Get high and low word from the RG arrays */
        lo_rg = p->yuv_rg_l[index];
        hi = p->yuv_rg_h[index];

        /* Add high and low words from the B arrays */
        lo = lo_rg + p->yuv_b_l[i2];
        hi += p->yuv_b_h[i2];

        /* Test for carry */
        if (lo < lo_rg) {
            /* there was a carry! */
            hi++;

            /* Store results */
            *ua++ = hi;
            *yar++ = lo>>12L;
            *var++ = hi>>21L;
        } else {
            /* Store results */
            *ua++ = hi;
            *yar++ = lo>>12L;
            *var++ = hi>>21L;
        }
    }
}
```

LISTING 2

```
/* Prototypes */

void *RGBtoYUVInit(void);

void RGBtoYUV(unsigned char *ra, unsigned char *ga,
    unsigned char *ba, unsigned char *ya,
    signed char *ua, signed char *va,
    unsigned long numpix, void *pd);


/* Typedefs to tame the C language. No compiler switches, because short always
 * seems to be 16-bit... */
typedef signed char signed8;
typedef unsigned char unsigned8;
typedef signed short signed16;
typedef unsigned short unsigned16;
typedef unsigned long unsigned32;
typedef signed long signed32;

/* Data structure for the parallel-add algorithms */
typedef struct rgb_yuv_data {
    unsigned32  yuv_rg_l[65536L];
    unsigned32  yuv_rg_h[65536L];
    unsigned32  yuv_b_l[256];
    unsigned32  yuv_b_h[256];
} rgb_yuv_data;
```

Transform matrix: Determines the orientation of the RGB color cube within YUV space, and the relative intensities of R, G, and B.
NOTE: Whenever the matrix is changed, error analysis has to be done to determine if 20 bits is still enough accuracy to determine the rounding direction of results!

```
signed32 m1[] = {
  2508194L,  4924113L,   956301L,
 -1415459L, -2778845L,  4194304L,
  4194304L, -3512206L,  -682098L };
```

RGBtoYUVInit

```
/* Parallel addition, 64-bit math version.
 * Hi format is 00CVVVVVVVVVvvvvvvvvvvvvCUUUUUUUUU
 * Lo format is uuuuuuuuuuuuYYYYYYYYyyyyyyyyyyyy
 */
void *RGBtoYUVInit()
{
    rgb_yuv_data  *p;
    Handle        h;
    OSErr         err;
    unsigned16    r, g, b;
    signed32      index;
    signed32      yl, ul, vl;
    unsigned32    yi;
    signed32      ui, vi;
    unsigned32    round_adjust = 0x7fe;
    unsigned32    lo12 = 0x0fffL;
    unsigned32    lo20 = 0xfffffL;
    unsigned32    datasize = sizeof(rgb_yuv_data);

    h = TempNewHandle(datasize, &err);
    HLock(h);
    p = *((rgb_yuv_data **) h);

    for (r=0; r<256; r++) {
        for (g=0; g<256; g++) {
            yl = m1[0] * ((signed32) r)
                       + m1[1] * ((signed32) g);
            ul = m1[3] * ((signed32) r)
                       + m1[4] * ((signed32) g);
            vl = m1[6] * ((signed32) r)
                       + m1[7] * ((signed32) g);

            yi = (yl + 1024)>>11L; yi += round_adjust;
            ui = (ul + 1024)>>11L; ui += round_adjust;
            vi = (vl + 1024)>>11L; vi += round_adjust;

            index = (((long)r)<<8L) | ((long)g);

            (p->yuv_rg_h)[index] =
                ((vi&lo20)<<9L) | ((ui&lo20)>>12L);

            (p->yuv_rg_l)[index] = ((ui&lo12)<<20L) | yi;
        }
    }

    for (b=0; b<256; b++) {
        yl = m1[2] * ((signed32) b);
        ul = m1[5] * ((signed32) b);
        vl = m1[8] * ((signed32) b);
```

```
        yi = (yl + 1024)>>11L;
        ui = (ul + 1024)>>11L;
        vi = (vl + 1024)>>11L;

        index = b;
        p->yuv_b_h[index] =
                    ((vi&lo20)<<9L) | ((ui&lo20)>>12L);

        p->yuv_b_l[index] = ((ui&lo12)<<20L) + yi;
    }

    return ((void *)p);
}
```

RGBtoYUV for 64-bit math

RGBtoYUV for 64-bit math version.
It's hard to read because the instructions were reordered to minimize pipeline stalls from result dependencies on the '040.

```
void RGBtoYUV(unsigned char *ra, unsigned char *ga,
    unsigned char *ba, unsigned char *ya,
    signed char *ua, signed char *va,
    unsigned long numpix, void *pd)
{
    register signed32       index;
    register signed32       i2;
             unsigned32     i;
    register unsigned32     hi, lo_rg, lo;
    register unsigned char *yar = ya;
    register signed char   *var = va;
    register unsigned8      *p;

    for(i=0; i<numpix; i++) {
        p = (unsigned8 *) pd;
        index = ((long)(*ra));          /* Compute indexes */
        i2 = ((long)(*ba));
        index <<= 8L;
        index += ((long)(*ga));
        i2 <<= 2L;
        index <<= 2L;
        ga++;                           /* Increment src ptrs */
        p += index;
        lo_rg = *((unsigned32 *)(p));   /* yuv_rg, low */
        p += 262144L;
        ra++;
        hi = *((unsigned32 *)(p));      /* yuv_rg, high */
        index -= i2;
        p += 262144L;
        ba++;
        p -= index;
        lo = lo_rg + *((unsigned32 *)(p));
        hi += *((unsigned32 *)(p+1024L));
        if (lo < lo_rg) {
            hi++;                       /* there was a carry! */
            lo >>= 6L;                  /* Store the results */
            *ua = hi;
            lo >>= 6L;
            *yar++ = lo;
            hi >>= 21L;
            ua++;
            *var++ = hi;
        } else {
            lo >>= 6L;                  /* Store the results */
            *ua = hi;
            lo >>= 6L;
            *yar++ = lo;
            hi >>= 21L;
            ua++;
            *var++ = hi;
        }
    }
}
```

By Chris Espinosa, Apple Computer, MacTech Magazine Regular Contributor

Think Like a Moviemaker

You may have a roster that really does look like movie credits...

If your development shop is more than just yourself, you probably have some separation of function among the people working on a project. Usually there's a separation between development and testing; if you're big enough, documentation is a third branch. Some organizations last a long time without really breaking the organization up into more parts than this.

But as your development team gets bigger, you may need to break the developers into teams. Often one team is set to work on the core technology and another on the interface, or one team does user interface work and another does system-level stuff. If your project gets very large, you may have more than two teams working on different modules of the project.

And everybody knows that as you increase the number of teams, the development time increases geometrically, because of the added burden of communicating among the teams. This stage of growth can be fatal for small companies. Things stop working like they used to. The kind of easy collaboration that happened over cubicle walls doesn't happen from hall to hall, or building to building.

And unless carefully managed, the teams can go in different directions, duplicate each others' work, or leave holes in the product big enough for competitors to drive through.

And nobody's making it simpler for you. With each year the requirements on you go up: you need experts in cross-platform development, better user interface designers, and multimedia mavens to keep up with new technologies and user demands. At some point the credits in the About... box start looking like the credits at the end of a movie.

Take a cue from that. Movies are pretty complex creative processes, taking dozens of people months or years to create. And while the analogy breaks down pretty easily (a bug in a movie doesn't generate tech support calls, for example), there's a lot to be learned from the way a studio organizes a movie production.

And I'm not talking about big-budget epics here. Thousands of made-for-TV movies, direct-to-rental releases, and minor studio pictures are made each year, and there's a lot of consistency in the way they're put together from studio to studio.

First of all, somebody's in charge: there's a producer responsible for bringing it to market and managing the investment, and a director responsible for the creativity and quality of the picture, and managing the people. The director may work on only one picture at a time. How different this is from a software company of moderate size! In such companies, people probably report up through functional organizations that work across products, and there's no equivalent of a "producer" until you get to a division VP, who doesn't have enough personal involvement to make authoritative decisions. To correct this, you may have a project leader to function as a "director," but without the management authority to tell people what to do.

Next (and most distressing for people who want job security in high-tech): most people who work on a picture are independent contractors. There was a time when actors, directors, and cinematographers were firmly attached to one studio, but nowadays they float from studio to studio, picture to picture. Even sequels and serials are made by different teams of people. This is easy in Hollywood, where tools, equipment, and raw materials are pretty much the same from job to job. The training time currently needed to learn and understand a large body of C++ source code puts a damper on hiring journeymen programmers for a specific job, and the desire to not have your trade secrets go to the competition is a reason to

keep your engineers around.

But I predict that over time that will change, as less and less coding is required to develop saleable software, and more and more the job of creating applications becomes one of fitting new components into an existing structure, or redefining the relationships among components. This is already happening in other areas of technology (like VLSI design, where designers rarely work at the gate level, much less the transistor); that software programmers still write code a line at a time is as crazy as trying to make movies with a still camera.

So in the organization of an application team, you may have a roster that really does look like movie credits: a producer and a director; a couple of assistant producers and directors to manage the relationship with the platform vendor; an overall architect (like a scriptwriter) and a few lead programmers (the lead actors); a supporting cast of programmers and testers, brought in for the job; an interface stylist, a sound editor, a technical crew doing tools and integration; and of course second unit to port the application to a different platform. After the project, the work gets filed in a good source repository, the leads may stay with you, but the majority of the team moves on to other studios and other jobs.

Next month: making the movie...

THINK TOP 10

By Mark B. Baldwin and Craig Connor, Symantec Technical Support

SYMANTEC.

This is a monthly column written by Symantec's Technical Support Engineers intended to provide you with information on Symantec products. Each month we cover either a specific application of tools or a "Q&A" list.

Q. *Can pointers to base classes be cast as pointers to derived classes?*

A. Yes. Run-Time Type Identification (RTTI), allows the safe casting of base class pointers to derived classes. If the cast is not allowed, a NULL pointer is returned.

Q. *How come I get an error when I declare a variable of any type with the name 'v' or 'h'?*

A. There's an enumeration in Types.h that looks like:

```
enum { h, v };
```

which clutters the namespace for both h and v.

Q. *Using TCL, when I create a series of CEditText panes in a window, why are they placed at seemingly random places on the screen?*

A. The coordinates that you are setting for your CEditText object are not being interpreted as window coordinates. Have your object call FrameToWindR(Rect *windowCoordinates). This will adjust the frame of the pane to the correct position on the screen.

Q. *Why doesn't the example in the manual for CArrayIterator work? I get the following:*

```
File "CPtrArrayIterator_myClass.cp"; Line 7
Error:   'gAncestors' is not a member of struct
'CPtrArrayIterator<myClass>'

File "CPtrArrayIterator_myClass.cp"; Line 9
Error:   unable to open input file 'CPtrArrayIterator.tem'
```

A. The example in the manual for CArrayIterator is incorrect. 1) Remove the line for TCL_DEFINE_CLASS. You do this because the base class for CArrayIterator is not an RTTI class. 2) Remove the #include CArrayIterator.tem. This file does not get created because you are no longer calling TCL_DEFINE_CLASS.

Q. *How do I change the foreground/ background color of a CStaticText?*

A. Instantiate a pointer to your CStaticText object, or if you are using VA, find where the object pointer is created in x_CMain. Then use the following formula:

```
((*CColorTextEnvirons)myCStaticText->itsEnvirons)
  ->SetColorInfo(*RGBColor forecolor, *RGBColor backcolor);
```

Notice we cast from a base to a derived class CEnvrirons to a derived class CColorTextEnvirons.

Q. *How do I read in a CBitMapPane?*

A. Try the following code segment. It reads a bitmap from a file and assigns it to a CBitMapPane object.

```
CPNTGFile   *theFile = NULL;
SFTypeList  myList;
SFReply theReply;
Point   where;

where.h=120;  where.v=190;   // SF dialog window position

myList[0]='PNTG';

SFGetFile(where,0,NULL,1,myList,0,&theReply);

if(theReply.good)
{
    theFile = new CPNTGFile;        // Make a File object and read the
    theFile->SFSpecify(&theReply);// data into a new BitMap
```

C/C++ without Object Master

Introducing a powerful new way of looking at code development.

Open your eyes to Object Master™, the most innovative programming tool available on the market today. With its powerful editors and intuitive windows, Object Master gives you the unsurpassed freedom to develop code quickly and accurately.

A Real Eye Opener

Use Object Master's unlimited number of browser windows to access code components and display their definitions, ready for editing. All changes you make in the browser windows are automatically displayed throughout the environment, consistently ensuring current and accurate code. With the browser windows, you can also view a class list displaying the hierarchical relationships between classes, and use pre-made templates to create classes and methods.

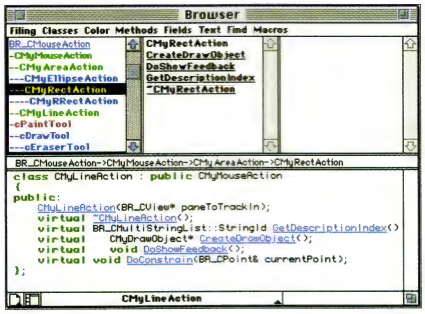
Picture Perfect Code

While the browser windows let you edit specific pieces of code, Object Master's File editor gives you a full-file view of your code, allowing you to quickly perform universal edits.

To help you identify important pieces of code easily, Object Master color codes and formats language elements. With a single keystroke, Object Master will look up parameters for methods or functions contained in the project and paste them directly into your source code.

Improved Insight

Don't worry about the physical location of your code—Object Master parses all files and maintains a data dictionary of project components. The dynamic environment updates your entire project automatically as you edit without compilation!



True browser windows let you see code like never before.

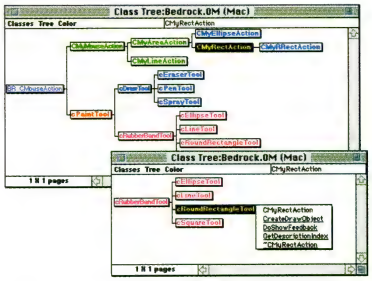
Object Master provides you with all the editing and navigational capabilities you require to write code productively. For example, the Class Tree window graphically displays your project's class hierarchy and allows you to expand and collapse "branches" and open multiple windows displaying specific code components.

Power Macintosh Support

ACI offers two versions of this outstanding programming tool: Object Master and Object Master UNIVERSAL. Both run on the traditional 68K-based Macintosh computer and the new Power Macintosh, taking full advantage of the features of each platform.

Object Master supports C and C++ and works seamlessly with Symantec's THINK Project Manager and Metrowerk's CodeWarrior. Separate versions are available for the 68K-based Macintosh and the new Power Macintosh computers.

Object Master UNIVERSAL supports C, C++, Pascal, Modula-2, and all major compilation systems. It can be installed on both the 68K-based Macintosh and the new Power Macintosh.



"...Object Master pays for itself in a week, even at suggested retail price." –Macworld Magazine

See Object Master for Yourself

Call **(800) 384-0010**, and we'll send you a free demo disk of Object Master—because seeing really is believing.

ACI US Inc. 20883 Stevens Creek Blvd., Cupertino, CA 95014
Tel. 1 408 252 4444. Fax 1 408 252 0831. AppleLink D4444

```
    theFile->Open(fsRdWrPerm);
    theBitMap = ((CPNTGFile *)itsFile)->ReadNewBitMap(TRUE);

    myBitMapPane->SetBitMap(theBitMap);
}
```

Q. *How do I use VA to make CIconButtons work like radio buttons? It works when I do a Try Out, but then the code is not generated.*

A. Version 7.0.3 fixes this problem. Download the 7.0.3 Patch from an on-line service.

Q. *How do I make a PICTGrid in a Window using VA?*

A. Create the window you want to use. Temporarily create a tear-off menu view, and copy the PICTGrid from the menu, and paste it into your window. Then delete the tear-off menu.

Q. *Everytime I create a VA project it takes a very long time to compile. How can I reduce the compile time?*

A. Open the "Project Models:Visual Architect:@1.1" file, and compile it. From now on, the TCL files will be compiled upon the creation of a new VA project.

Q. *How do I add a new project type to the project models that appear in the New Project Dialog?*

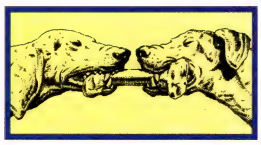
A. Create a new project, include the files and libraries you want, compile it, and copy the whole folder over to the Project Models folder. Name the project file "@1.1". The "@1" will be replaced with the project name you specify. If you have trouble, take a look at the existing project models to see the naming convention for the project files.

Special Thanks To

Steve Howard, Michael Hopkins, Colen Garoutte-Carson, Rick Hartmann, Kevin Irlen, Yuen Li, Celso Barriga, Scott Shurr, and Chris Prinos, et al.

To receive information on any products advertised in this issue, send your request via Internet: productinfo@xplain.com

PROGRAMMER'S CHALLENGE



By Mike Scanlin, Mountain View, CA

How Long Will It Take?

All of the programmer challenges during the last couple of years have focused on optimizing algorithms and implementations. This month we have something a little different. We're going to tackle one of the hardest problems every software engineer has to deal with. No, I'm not talking about some weird memory model compatibility problem on DOS machines; I'm talking about scheduling. Specifically, estimating how long a particular software project will take. We all know it's hard for us subjective humans to do this task accurately but maybe one of you clever readers can come up with an algorithmic way to estimate a project. And, you can even decide what your parameters will be.

Here are some example parameters you might want to use to describe the software task at hand:

version = the number of major versions of this product that have already shipped (if you were working on System 8 then this number would be 7, for a new project it would be zero)

features = estimated number of major features that need to be implemented (for a text-based project the following are examples of major features: spell checking, printing, styles, find/replace, footnotes)

engSkilled = the number of very competent engineers working on the project (more than 5 years experience on the relevant platform using the relevant tools)

engNewGuys = the number of unskilled or relatively junior engineers working on the project (less than 2 years experience)

marketing = the number of full-time marketing folks working on the project

uiPeople = the number of people who have at least some decision making authority about the user interface of the project

qaPeople = the number of trained in-house testers assigned to the project

betaTesters = the number of out-of-house user testers using the product at least a month before code freeze

meetings = average number of meetings per engineer per week during the course of the project

love = a number from 1 to 5 describing how well the team members like each other (5 means everyone gets along great, respects and trusts each other; 1 means there are problems affecting work between several members).

linesC = estimated number of lines of code (of C)

objectKB = estimated executable size, in KB

mpw (boolean) = true if using MPW for compiling the

THE RULES

Here's how it works: Each month we present a different programming challenge here. First, you write some code that solves the challenge. Second, optimize your code (a lot). Then, submit your solution to MacTech Magazine (formerly MacTutor). We choose a winner based on code correctness, speed, size and elegance (in that order of importance) as well as the postmark of the answer. In the event of multiple equally desirable solutions, one winner will be chosen at random (with honorable mention, but no prize, given to the runners up). The prize for the best solution each month is $50 and a limited edition "The Winner! MacTech Magazine Programming Challenge" T-shirt (not available in stores).

In order to make fair comparisons between solutions, all solutions must be in ANSI compatible C (i.e., don't use Think's Object extensions). Use only pure C code. We will disqualify any entries with any assembly in them (except for those challenges specifically stated to be in assembly). You may call any routine in the Macintosh toolbox you want (e.g., it doesn't matter if you use NewPtr instead of malloc). We test entries with the FPU and 68020 flags turned *off* in THINK C. We time routines with the latest version of THINK C (with "ANSI Settings", "Honor 'register' first", and "Use Global Optimizer" turned on), so beware if you optimize for a different C compiler. **Limit your code to 60 characters wide. This helps us deal with e-mail gateways and simplifies page layout.**

We publish the solution and winners for this month's Programmers' Challenge in the issue two months later. All submissions must be ***received*** *by* the 10th day of the month printed on the front of this issue.

Mark solutions "Attn: Programmers' Challenge Solution" and send them via e-mail – *Internet* progchallenge@xplain.com, *AppleLink* MT.PROGCHAL, *CompuServe* 71552,174 and *America Online* MT PRGCHAL. Include the solution, all related files, and your contact information. If you send via snail mail, please send a disk with those items on it; see "How to Contact Xplain Corporation" on page 2.

MacTech Magazine reserves the right to publish any solution entered in the Programming Challenge of the Month. Authors grant MacTech Magazine the non-exclusive right to publish entries without limitation upon submission of each entry. Authors retain copyrights for the code.

dtF The Relational Database System

What is dtF...

dtF is a true relational database management system for C/C++ application development. dtF features SQL, full transaction control, error recovery and client-server architecture.

dtF is royalty free...

Create and distribute as many applications as you like, royalty free!

dtF is fast...

When it comes to performance dtF is in a class all its own. dtF utilizes a proprietary query optimization and caching scheme to obtain unparalleled performance.

dtF is efficient...

dtF was developed by Macintosh developers for Macintosh developers. Applications developed with dtF will be compact in both single and multi user form. Perfect RDBMS for Powerbook applications.

dtF is SQL...

dtF supports an efficient subset of ISO standard SQL that is optimized for speed and ease of use.

dtF is safe...

Full transaction control and error recovery guarantee maximum data protection even after sudden system crashes. dtF databases are compressed and encrypted to protect against all unauthorized access, even disk editors.

dtF is easy...

Integrated data dictionary, security, automatic index selection, query optimization, deadlock detection and error recovery allow you concentrate on your application.

dtF is true client server...

dtF client server architecture insures efficient use of network bandwidth and optimum data security. dtF will not bog down your network like systems that use file sharing.

dtF single user...

Client server applications will not be stranded on the LAN. Relink your application with dtF single user and you will have a no compromises high performance stand alone.

dtF is for you!

For maximum performance, realistic licensing policy and reasonable pricing you can not beat dtF. No static preprocessors, only a dynamic, fully featured SQL. The C/C++ API is identical and fully portable over all supported platforms in both single and multi-user environments. dtF is ideal for use with TCL and MacApp.

dtF Platforms and Compilers...

Available for Macintosh System 7.x and native Power PC. dtF Server requires a 68020 or better. MPW C/C++ and Symantec C/C++ 5.x and higher are both supported.

dtF Evaluation	$129
dtF Macintosh SDK	$695
dtF LAN Macintosh SDK*	$1595
dtF Server	$1295

dtF Americas, Inc.
12545 Olive Blvd, Suite 130
St. Louis, MO 63141
800.DTF.1790 Voice
314.530.1697 Fax
AppleLink: DTF.AMERICA

dtF is also available for DOS, Windows, OS/2 and several flavors of UNIX.

base project

thinkC (boolean) = true if using Think C for compiling the base project

appFramework (boolean) = true if using someone else's application framework or class library (like TCL or MacApp)

systemCost = average number of dollars per engineer spent on engineering development hardware (do the engineers have ample CPU, RAM, disk space, etc.?)

personalMoney (boolean) = true if the engineer's are financing this project at least partially with their own money

bonusMoney (boolean) = true if there is a meaningful bonus for the team if the project is done on time

food (boolean) = company provides adequate in-house food or, there is at least one restaurant that delivers food 24 hours a day

dew (boolean) = true if free Mountain Dew is available in-house

toys = average number of toys per engineer

netnews (boolean) = true if netnews is available

email (boolean) = true if e-mail is available

The prototype of the function you write starts like this:

```
unsigned short SoftwareTimeEstimate(...);
```

and it's up to you to fill in the list of parameters (including their types, probable ranges and maybe an example of each). You can choose from the list above or make up your own. The return value is the estimated total time (in calendar days) that the software project will take to make the golden master disk (but it does not include time for printing the manual, disk-duplicating, shipping, etc.).

In order to limit the scope of the project a little bit, let's assume the project is a typical general-purpose Macintosh application. It could center around graphics, text, spreadsheet, database, communications, etc (nothing too vertical or specialized, though). You can give as much or as little weight to any of the parameters as you like. Your parameters should not be too specific because they need to work for a fairly broad selection of software projects ranging from version 1.0 of a new word processor to version 8.0 of an existing spreadsheet.

Unlike normal challenges, this challenge will not be judged on speed or code size. Instead each entry will be graded by a panel of at least three judges who will give a numerical score to each entry in each of three categories: (1) realistic (i.e. someone *might* actually be able to use it and get a *somewhat* usable number out of it), (2) documented assumptions, opinions and coefficients (explain at least a little bit how each variable, ratio and coefficient affects the final answer you produce; use lots of `#defines` so if someone disagrees with you they can redefine your coefficients and recompile to their taste) and (3) humor (let's not take this thing too seriously; the winning solution should be fun to read and maybe have a silly parameter or two). The judges' subjective scores will be totalled and the highest overall point total will win.

If you want to earn extra credit points for your entry then you can also submit your list of *Top 5 Excuses Why This Project Is Late* that you might give to management (who may or may not be technically impaired) once your deadline has passed. They can be meaningful excuses ("we need more equipment/people"), they can be actual excuses you've used in the past ("my disk crashed; we have no backup") and/or they can be things you'd like to use but probably won't for fear of reprisals ("If you'd stop asking me every 5 minutes when it will be done then it will be done a hell of a lot sooner!"). For each excuse the judges like we'll add a few points to your overall score.

One last request: Please don't go completely wack-o in terms of the length of your entry. The winning solution should fit on 2 to 6 standard MacTech code-listing type pages. This may limit the accuracy of your entry a bit (depending on how detailed you want to be) but you'll just have to live with that and concentrate on the most important parameters first. As always, e-mail me if you have any questions.

Two Months Ago Winner

Congratulations to **Dave Darrah** (Lansdale, PA) for his winning entry in the DumpBytes challenge. Despite a bug in Think C (which he was able to identify and work around) Dave was able to dump bytes faster than anyone else and, he was able to do it with relatively little code and lookup table data (4th smallest entry overall). Nice job!

Here are the times and code+data sizes for each entry. The code+data size represents the code size plus the size of the static data (i.e. lookup tables). Numbers in parens after a person's name indicate how many times that person has finished in the top 5 places of all previous Programmer Challenges, not including this one:

Name	time	code+data
Dave Darrah	64	1036
Ernst Munter (2)	75	3540
Bob Boonstra (11)	77	1514
N. Liber, I. Phillips (1)	83	6208
Ted DiSilvestre	86	1206
Allen Stenger (7)	87	2152
Kevin Cutts (3)	87	2286
Larry Landry (3)	103	1504
Steve Israelson (1)	128	714
Tom Elwertowski (1)	148	658
Mark Chavira	240	1514
Paul Stankiewicz	3270	422

The bug that Dave uncovered in Think C (and which I was able to reproduce) has to do with a static table of 512 chars. The very last entry (511th, zero-based) was not getting initialized to the value the auto-initializer declared it as. Dave was able to work around this bug by manually setting the last entry to the proper value early on in his code.

While reproducing this bug I noticed that if I increased the

size of the table to more than 512 chars then nothing after the 510th entry (zero-based) was initialized. I then thought of breaking the long quoted string into several smaller strings and that did indeed fix the problem. So, it appears there is a limitation in Think C of 511 bytes for the length of a quoted string. If you need more you should split it up into pieces like this:

```
static char myTable[] = \
"0123456789"\
"0123456789"\
...
"0123456789";
```

(wouldn't it be nice if the compiler informed you of its limits if and when you went beyond them...)

BlockMoveData

I saw something encouraging on AppleLink recently that readers of this column are sure to appreciate. Looks like Apple has finally accepted the idea that making BlockMove clear the instruction cache *every time it's called* was not efficient (it's only necessary when moving executable code). They have finally put an official interface on something I've been asking them to do for a long time: BlockMoveData. It's just like BlockMove but it doesn't do any cache flushing when it's done. Craig Prouse of Apple says, "It's only implemented in the $077D ROMs as found in the Quadra 840AV and Centris 660AV." (I suspect it's also implemented on any newer ROMs, too...)

To use it all you have to do is set a bit in the trap word. Normally BlockMove is 0xA02E but, if you use 0xA22E instead then you'll get the new BlockMoveData for those ROMs where it's implemented (and you'll get regular old BlockMove on ROMs where it's not). So, unless you're moving executable code you should be using BlockMoveData for all your moves. Thanks Apple! And thanks Craig for posting this!

Here's Dave's winning solution:

Dump Bytes

Dave Darrah, Lansdale, PA

```
#include <string.h>

typedef unsigned short *  usp;

// For unknown reasons, Think generates more efficient code for the critical
// *(usp)outputText = aTablePtr[byte] instructions when
// register coloring is off. Go figure.

#pragma options (honor_register, !assign_registers, !gopt_coloring)

// Address registers are used for:
//    the "inputBytes" pointer, passed parameter.
//    the "outputText" pointer, passed parameter.
//    the "outputTextA" pointer, which points to where the ASCII representation goes.

// Data registers are used for:
//    "space", a holder of a space.
//    "byte", a temp area that holds a value we want to "burst" to ascii.
//    "eCntr, gCntr", counters used for two loops.
//    Pointer to "aTable": "aTablePtr". Think assigns the last data reg to it.

unsigned short    DumpBytes(inputBytes,
                  outputText,
                  numInputBytes,
                  maxOutputBytes,
                  width,
                  grouping)

  register Ptr         inputBytes;
  register Ptr         outputText;
       unsigned short  numInputBytes;
       unsigned short  maxOutputBytes;
       unsigned short  width;
       unsigned short  grouping;


  register Byte      space=' ';
  register Byte      byte;
  register unsigned short  eCntr,gCntr;

  register Ptr       outputTextA;

  unsigned short     dispValue=0;
// This is the hex value of what's printed at the beginning of each line.

  unsigned short     groupsPerLine,lineLength,extras,
                     numberOfLines,asciiOffset,lastLineLength;

  Boolean            truncated=FALSE;
  Ptr                saveOutputText=outputText;

// 256 entry (512 byte) ascii table. This table will be indexed by the byte value,
// to return the two-char entry that is the character representation of that byte.
// See note later about how Think generates this table.

  static char      aTable[] = "\
000102030405060708090A0B0C0D0E0F\
101112131415161718191A1B1C1D1E1F\
202122232425262728292A2B2C2D2E2F\
303132333435363738393A3B3C3D3E3F\
404142434445464748494A4B4C4D4E4F\
505152535455565758595A5B5C5D5E5F\
606162636465666768696A6B6C6D6E6F\
707172737475767778797A7B7C7D7E7F\
808182838485868788898A8B8C8D8E8F\
909192939495969798999A9B9C9D9E9F\
A0A1A2A3A4A5A6A7A8A9AAABACADAEAF\
B0B1B2B3B4B5B6B7B8B9BABBBCBDBEBF\
C0C1C2C3C4C5C6C7C8C9CACBCCCDCECF\
D0D1D2D3D4D5D6D7D8D9DADBDCDDDEDF\
E0E1E2E3E4E5E6E7E8E9EAEBECEDEEEF\
F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF";

  register usp aTablePtr=(usp)&aTable;
  // Think gives aTablePtr a data register, and it actually helps!
```

Initialize and preflight

```
// That's it for local variables, first, let's initialize a few variables and preflight the
output length.

// Because of some flukie I haven't been able to scope out (a possible Think bug?), the
last value (the right "F" in "FF") of aTable is hex 0. Why? Don't know. Oh well, let's just
roll with the punch.

  aTable[511] = 'F';

// Calculate number of groups per line.

  groupsPerLine = width/grouping;

// Calculate output line length.

  lineLength = width*3 + groupsPerLine + 7;
        // 2 for hex representation, 1 for ascii;
        // 1 for each space that follows a group;
        // 4 for disp, a colon, space and return.

// Calculate offset from outputText where ascii goes.

  asciiOffset = lineLength - width - 1;
```

```
// Calculate the number of full lines of output.
   numberOfLines = numInputBytes/width;
// Calculate the number of bytes left over after all complete lines are done.
   extras = numInputBytes % width;
// Calculate the number of bytes this line takes.
   lastLineLength = asciiOffset + extras;
// Reduce numberOfLines if output would run past maxOutputBytes.
// Just dump the number of full lines that fit in maxOutputBytes.
   if ( (lineLength * (unsigned long)numberOfLines) +
        lastLineLength  > maxOutputBytes)  {
     numberOfLines = maxOutputBytes/lineLength;
     extras = 0;
     truncated = TRUE;
   }
```

```
// Initialization and preflighting done.  It's time to process the input.
   numberOfLines++;
   while (--numberOfLines)  { // Do each full line.
     outputTextA = outputText+asciiOffset;
     // Displacement value goes first.
     byte = dispValue>>8;      // left byte of disp
     *(usp)outputText = aTablePtr[byte];
     outputText += 2;
     byte = dispValue;         // right byte of disp
     *(usp)outputText = aTablePtr[byte];
     outputText += 2;
     *(usp)outputText = ': ';
     outputText += 2;
     dispValue += width;
     // Now do "groupsPerLine" sets of hex expansions.
     eCntr = groupsPerLine;
     do {                      // each of the "groupsPerLine" groups.
       gCntr = grouping;
       do {                    // each of the "grouping" bytes.
         byte = *inputBytes++;
         *(usp)outputText = aTablePtr[byte];
         outputText += 2;
         // Do the ascii.
         if (byte < space || byte > 0x7E)
           *outputTextA++ = '.';
         else
           *outputTextA++ = byte;
       } while (--gCntr);      // End "grouping bytes" loop
       *outputText++ = space;  // Space after each group
     } while (--eCntr);        // End "groups per line" loop
     *outputTextA++ = '\r';
     // Point to beginning of next line.
```

```
    outputText = outputTextA;

  }     // End of "lines" loop.

// Now to worry, if necessary, about dribble left over.
// A lot of code duplication, but what the hey!

  if ( extras )  {
    // Space output line.
    memset(outputText,space,lineLength);

    outputTextA = outputText+asciiOffset;

    // Displacement value goes first.
    byte = dispValue>>8;    // left byte of disp

    *(usp)outputText = aTablePtr[byte];
    outputText += 2;

    byte = dispValue;       // right byte of disp

    *(usp)outputText = aTablePtr[byte];
    outputText += 2;

    *outputText = ':';      // space is already there.
    outputText += 2;

    // Now do "groupsPerLine" sets of hex expansions.
    eCntr = groupsPerLine;
    do {            // each of the "groupsPerLine" groups.

      gCntr = grouping;
      do {          // each of the "grouping" bytes.

        byte = *inputBytes++;

        *(usp)outputText = aTablePtr[byte];
        outputText += 2;

        // Do the ascii.
        if (byte < space || byte > 0x7E)
          *outputTextA++ = '.';
        else
          *outputTextA++ = byte;

        if (!--extras)
          goto AllDone;    // Nasty termination when we've done 'em all.

      } while (--gCntr);   // End "bytes in group" loop

      // skip past space after each group.
      *outputText++;

    } while (--eCntr);     // End of groups per line loop.

  }  // end if

  AllDone:

  if (truncated)
    return 0;
  else
    return(outputTextA - saveOutputText);
}
```

By Mike Scanlin, Mountain View, CA

Scanlin on Books

POWER AND POWERPC

By Weiss and Smith
Morgan Kaufmann Publishers, Inc. 1994.
ISBN 1-55860-279-8.
408 pages (hardback).

If you are working on learning PowerPC assembly language, and you want a good, solid technical information that gives you both a base to start with and a reference to return to once you've started going, *POWER and PowerPC* delivers both. It is a very complete look at the POWER and PowerPC architectures and contains information that would be of interest to anyone who really wants to delve into PowerPC 601 programming.

The book covers three main areas: (1) the POWER architecture, (2) the first two POWER implementations, the POWER1 and POWER2, used by IBM in the RS/6000, and (3) the PowerPC architecture and PowerPC 601 implementation. There are other areas which are interesting but which probably aren't as important to most Macintosh programmers, including: A comparison of the POWER and PowerPC architectures, a comparison of the PowerPC 601 and DEC Alpha 21064, and the IEEE 754 Floating-Point standard.

The best part of this book, in my Mac-centric opinion, are Chapters 7-9, which describe the 601 in detail. It starts off with a discussion of the instruction formats and goes on to show how some of the less-obvious instructions work (like rotate with mask). It then goes on to explain the 601's pipelines, branch processing and caches. Within each of these discussions the examples are clearly illustrated with sample code fragments. You'll be able to see where and why pipeline stalls occur (and what you can do to avoid them in some cases), how to optimize your branches and exactly how the combined instruction and data cache works. Understanding these issues is a key part of being able to optimize for the 601 when you need to (in addition to helping you identify why certain code fragments run slower than you would expect).

This book is not a tutorial on how to program in PowerPC assembly language. It is, however, one of those rare technical books that is a pleasure to read for all the right reasons: the examples are clear, the examples are worth studying, the authors know their stuff and, it's presented in a neatly typeset and illustrated manner. I would recommend it to self-motivated people who want to start learning PowerPC assembly language programming or to anyone working in a high-level language who wants to know more about their underlying processor.

ZEN OF CODE OPTIMIZATION

By Michael Abrash
The Coriolis Group, Inc. 1994. ISBN 1-883577-03-9.
449 pages (soft cover, w/disk).

The guru who so many years ago brought us *The Zen of Assembly Language* has returned. He has now released a new and improved version of the ideas and examples contained in that sacred volume. And he has added new tricks for the latest Intel processors, the 486 and the Pentium.

Now, you are probably asking yourself "Why would I care about a bunch of optimization tricks on Intel processors when I'm a Mac fanatic?" The answer is because there is something for everyone in this book. Even if you ignore all of the Intel assembly code he presents (there is no 68K code at all) you can't help but be impressed with the methodology he used to determine the optimal instructions as well as the clear (and often times humorous) explanations of the finer points of assembly language programming. Mr. Abrash has a Zen-like understanding of the Intel processors and the environments they live in. He is also a gifted writer who can make an otherwise dry topic come to life. Even though I have personally

vowed never to write Intel assembly code, I thoroughly enjoyed both his former Zen book and this latest one.

The author sums up the book's essence rather well, "This book is the diary of a personal passion, my quest for ways to write the fastest possible software for IBM-compatible computers in C, C++, and assembly language. ... it is a journal of my exploration of the flexible mind in action (with, to be sure, a generous leavening of potent low-level optimization tricks)." This book is the summary of years of effort studying the subtle behavior of Intel processors. And most of it is presented in easy-to-read, story-like prose that is both fun to read and very educational.

The book starts off by giving us the Zen Timer; a little piece of code used throughout that gives you the most precise timings possible of your Intel code fragments. After all, you have to be able to measure your code accurately to know if your latest change really improved things or not.

The next couple of chapters teach you various low-level things you need to know to really optimize for the Intel processors, such as: the prefetch queue cycle-eater, dynamic RAM refresh cycle-eater, and the display adapter cycle-eater. The interaction of these cycle-eaters leads to some surprising results (like you can't trust the instruction times in the Intel manuals).

Once the basics are understood (and the reader has accepted "assume nothing; time everything") the book proceeds to apply that knowledge to some real-world problems. In particular, the Boyer-Moore string searching algorithm is studied and optimized. There are many examples of peephole optimizations (like fast multiplication by 5 or 9 with the LEA instruction). There are examples given on how to manipulate common data structures efficiently, such as linked lists.

The last few chapters are devoted to the Pentium. In addition to showing you how many of the 386 and 486 tricks (taught in earlier chapters) will break on the Pentium, there is an in-depth discussion of the Pentium and it's U-pipe and V-pipe and how to keep them both full most of the time. Sadly, like the 68K family, it is not possible to simultaneously optimize for all members of the Intel family.

As an optimizing assembly language programmer, I found it refreshing to find someone who is both a true master at assembly language programming and at the same time capable of making all the right trade-offs when coding in mixed C and assembly. I would recommend *Zen of Code Optimization* as a 10 on a scale of 10 if you are working on any Intel processor and I would give it an 8 out of 10 for anyone working on the Macintosh who is interested in writing high performance code.

JPEG: Still Image Data Compression Standard
By Pennebaker and Mitchell
Van Nostrand Reinhold. 1993. ISBN 0-442-01272-1.
638 pages (hardback).

Ever wonder how those color painting programs manage to store 10MB of pixels in a 1MB file? Well, most of them use JPEG (Joint Photographic Experts Group) image compression. If you've ever wanted to know how it works, or implement it yourself, then *JPEG: Still Image Data Compression Standard* is the book for you.

Written by two members of the JPEG standard committee, this book gives many of the hows and whys of JPEG that are not in the official JPEG specification (which is given in an appendix). There is a good chance that this book will tell you more than you really want to know about JPEG. It contains a LOT of information.

The book is written for everyone, from non-technical people to programmers to mathematicians. But don't let the inclusion of some non-technical info dissuade you – there is more technical and mathematical info here than you probably care to read. Each section is marked with one of three "technical difficulty" symbols: one for non-technical readers, one for people with intermediate technical skills and one for people with advanced technical skills who are either going to implement a JPEG engine or else just like hard math problems.

The beginning of the book goes over some basic imaging concepts for the uninitiated (such as low-pass filters and the difference between luminance and chrominance). It then introduces you to the Discrete Cosine Transform (DCT) that is the heart of JPEG. This discussion is very complete and certainly makes it clear how both one dimensional and two dimensional DCTs work (with good illustrations and examples). It also explains some of the 'blocking' effects you sometimes see with JPEG images.

After the DCT, the book goes on to explain the various JPEG modes of operation (sequential, progressive, lossless and hierarchical) and the syntax of the JPEG data stream. If you've read the spec and not been clear on any of those things then this book's discussion of them will clear them up for you (it certainly did for me).

Once you've run your image data through the DCT and quantized it, the last step of JPEG is to entropy encode the quantized values. JPEG allows two methods of entropy encoding: Huffman or arithmetic. The book spends ample time on both methods (several chapters, in fact, including one on probability estimation) and, depending on how much you like math, you'll come away either really confused or really understanding how it works. (The explanations are clear, but it's difficult material.)

The last part of the book gives comparisons of performance for the different kinds of JPEG compression, a list of JPEG applications and vendors, a history of JPEG, possible future directions of JPEG and a discussion of non-JPEG compression standards (JBIG, MPEG, fax).

This book is as complete as you could possibly want on the subject of JPEG image compression. It is a must-read for JPEG implementors and recommended reading for those people who like to understand how common algorithms work or who want to know more about imaging algorithms in general.

NEW APPLE TECHNOLOGY

By Glenn Andreas, Fridley, MN

Making MIDI Music

Using QuickTime 2.0 to make some music of your own...

Editor's Introduction – Glenn tried to use Apple docs to figure out how to generate MIDI music with QuickTime 2.0. He ran into the common insufficient documentation (what's that error number, again?) *roadblock, and dove in to figure out how things really operate. In doing so, he ran some risk of learning how to do things the wrong way, so we gave some good folks at Apple a crack at his article. While calling the approach a bit "hackish" because he defines his own headers, they say he pretty much got it right. While there are easier ways to put MIDI tunes into QuickTime movies and play them as background music, this article shows you how to drive from the API level. It's rather like using the Sound Manager with QuickTime.*

Two caveats – the flags in TuneStop are not implemented (the article suggests they are) and TuneResume doesn't.

Be sure to check out the real documentation when it comes available, and enjoy MusicTest in the meantime!

The Motivation

As I worked on my latest project, I wanted to have background music. I was originally planning on writing some sort of "auto mixing my own sound buffers from hell" sound manager hack, and while these are fun to write, it would take a great deal of resources to get it correct. Then I heard that QuickTime 2.0 would include the ability to play music, as music. "Sort of like MIDI" I heard. Call me a crazy, but I'd rather just make a few component manager calls rather than spend months writing my own music playing routines (besides having all my work done for me, QT 2.0 can play the music through an external MIDI device as well for even better quality with almost no system load). So, I got a beta of QT 2.0, and immediately dove into the "Macintosh Music Architecture" document. This was rather like diving in the shallow end of the pool. Given that the header code and the documentation didn't synch all that well, and assuming that the underlying code followed yet another convention, I did what any resourceful programmer would do – I dropped into MacsBug.

Fortunately for me, MoviePlayer was able to correctly play some of the sample music movies included on the beta CD. And, also fortunately for me, the whole music architecture is build uses the component manager (which, by the way for those who have never really looked at it, is really cool). So I set a breakpoint at the beginning of the "Tune Player Component", and watched every call made by QuickTime as it played music. And finally, after countless reboots, I was able to make my Mac play music from my own programs.

At this point, I saw this as an opportunity for a little fortune and glory. Given that the only existing document I had on how to play music was inaccurate, I figured that I would write this article explaining just what it will take for you to easily add background music to your current or future project. Note that

Glenn Andreas – Glenn started Mac programming in the Fall of 84 when he conned his boss into buying him a Lisa and the original IM. Since then, he's written the game *Theldrow*, worked for Palomar Software writing printer drivers, and done various freelance programming jobs, including a sped up version of the Stylewriter driver (which never shipped), and some work on the printing portions of Bedrock (which also never shipped). He's currently working a "day job" which involves hacking the BSD kernel, while trying to finish his latest game *Chimera* (which will hopefully ship one day).

this article is in no way official documentation, it is simply what I have discovered on how things works. I'll only present a subset of the routines available (see the header files for QuickTime 2.0 which appear in the August Developer CD for more details), and all examples will be based on what I've found in snooping around (so if I say that this parameter is zero, that means that I've always seen this parameter as zero, not that it has to be – but unless you like rebooting, you might want to leave it as zero). Also, I will attempt to avoid as much "music theory" as possible, since not only are there good books on this already, my knowledge is weak in that area, and I wouldn't want to provide misleading information. This will be in Pascal, and I'll simplify the header file so we don't have to include dozens of other files (because QuickTime wants Aliases which wants AppleTalk which wants, well, you get the idea). First, however, let's look at a quick bit of pseudo-code and see what it is we are going to do.

Overview

In order to play music, we need two important pieces of information – what notes to play, and what instruments to play them on. This first part is the body of the tune, while the second is what is called the header of the tune. In QuickTime, the tune header, along with some additional information, is stored in the media handler information (in the resource fork), while the tune body is stored in the actual data (on the data fork). For this article, we will store both in a single resource, starting with what the media handler information would be (as defined in `MusicDescription` record), with the tune body appended onto the end.

Here, then, are the basic steps we will use to play music:

- Create a tune player component
- Feed the music header the header from our resource
- Tell it to start playing the body from our resource
- Wait until it finishes
- Make sure to tell it to stop playing
- Dispose and clean up.

But before we get to the code, let's look how the music is stored (this is important, so don't skip ahead).

Storage of Musical Notes

Music is stored as a series of commands. Each command usually takes one longint, but can take two or more. Examples of these commands are to play a given note on a given instrument at a given pitch and given volume for a given duration, or to have a given instrument wait a given duration. And thanks to the "Time" part of QuickTime, multiple instruments can all be synched together, or their tempo can just as easily be changed (but for you QuickTime junkies, I will not be getting into time bases).

These commands are similar to MIDI commands, if you are familiar with them. If you aren't, by the end of this section you'll know that MIDI commands are similar to QuickTime music commands. These commands are tagged with what the command is in the high three or four bits, with the remaining bits (or following long word or words) providing the parameters. Note that all commands are multiples of four bytes long, so you can easily scan them as an array of LongInts.

Before we get to the commands themselves, we need to look at some of the parameters first. Almost all commands require a integer parameter to specify which instrument the command affects. An instrument is just that – a single instrument. QuickTime provides 30 some odd instruments to choose from. You can have more than one instrument playing in a given song (so you can have "Dueling -insert your favorite instrument here-"). Also, a single instrument can play more than one note at a time (being able to play a chord). Before playing the song, you tell the tune player component how many instruments there are and what they are, but we'll get into this later.

Other parameters are fairly self explanatory.

Volume (also called velocity, because it refers to how hard you strike the key on a keyboard), ranges from 0 (silent) to 63.

Duration is specified in units that are specific to the tune component (in our examples we use 1/600ths of a second). *Warning, some music theory follows:* Based on the default time units, and assuming 4/4 time (which means that a quarter note is 1/2 second long), below is a quick table of duration values and the length of notes they produce:

Units at 600/second	*Note produced*
75	sixteenth note
150	eight note
300	quarter note
600	half note
1200	whole note

Pitch, which ranges from 0 to 127, corresponds to the same values MIDI uses. Pitch ranges from C five octaves below middle C (0) to G five octaves above middle C (127). Middle C has the value of 60. A complete table of these values can be found on page 2-43 of the recently published *IM:Sound*. While it is possible to play what is called "microtonal values", which can be just about any pitch (they are represented with fixed point numbers), this is done via a different, much less convenient, interface.

One other thing to be aware of is that most commands have an "extended" form that usually takes two long words instead of one. This allows more bits for each parameter, but in all my "snooping" I've yet to come across anything that uses them. For the sake of simplicity, I'll also restrict myself to just the commands that are commonly used. In the commands below, I'll show all thirty two bits, with the most significant bit on the left, with a space between each four bits for easier reading.

Our first command is the "rest" command:

```
000- ---- dddd dddd dddd dddd dddd dddd
```

The 000 is the rest command. The next five bits would normally be for the instrument, but for the rest command are unused (and should be set to zero). Those are followed by twenty-four bits of duration d. This just instructs the instrument not to start any more commands until that time has passed.

The next command is the "note" command:

```
001i iiii pppp ppvv vvvv vddd dddd dddd
```

The 001 is the note command, the i's are five bits representing what instrument (0-31), and the d's are again duration (though for this command there are only 11 bits, so the value ranges from 0 to 2047, or in our examples, over 3 seconds). The six bits of p's are our pitch, again as a MIDI note value, and the seven bits of v's are the volume.

These two commands are enough to start playing music. However, if you wanted to play a scale, and you just issued eight note commands with increasing pitches, you would get one short polyphonic cacophony – all the notes would play at the same time, as a chord, rather than as eight sequential notes. To achieve the desired effect (each note in sequence), you need to issue eight note-rest command pairs, to allow time for one note to play before starting the next.

One final command in this example is needed – a command to have it stop. Without it, it will start playing whatever's in memory, and soon you'll be rebooting. The command to stop playing is the "marker" command:

```
011- ---- ssss ssss xxxx xxxx xxxx xxxx
```

The 011 is the marker command, the '-'s are unused (and should be set to zero). The eight bits of s are the marker event subtype, and the x's are the marker event value. The only values I've seen are zero for both, which is used to mark the end of the playing – all music command lists end with the value $60000000.

For the sake of completeness, here are the rest of the commands (though we will only be using one of them):

```
010i iiii cccc cccc xxxx xxxx xxxx xxxx
```

Control: i is instrument, c is controller, x is the value to set that controller to.

```
1011 iiii iiii iiii hhhh hhhh hhhh hhhh
10kk kkkk kkkk kkkk llll llll llll llll
```

Knob: i is instrument, k is what knob, h is the high word and l is the low word of what value to set that knob to.

```
1001 iiii iiii iiii pppp pppp pppp pppp
10-- ---- --vv vvvv vvvv vvvv vvvv vvvv
```

Extended note, with the same parameters as note.

```
1010 iiii iiii iiii cccc cccc cccc cccc
10-- ---- ---- ---- xxxx xxxx xxxx xxxx
```

Extended control, with the same parameters as control.

```
1111 iiii iiii iiii llll llll llll llll
....
10tt tttt tttt tttt llll llll llll llll
```

This is the general command, which isn't used in playing the music, but rather is used when you set up the tune player component. It is this command that is used to tell the player what instruments are what. The i's, of course, are the instrument, the t's represent one of the following:

1	Note request
2	Instrument
3	Flat Instrument
4	Part Name
5	Part Key

I've only seen "Note Request", so that is all that I will talk about. The l's represent the length of the entire command, including any general data. This value is in long words, and includes both long words of the command. Finally, between the two commands is some amount of additional data. Here's an example to help explain this – this is the command from the header portion of some music, and it instructs the tune player that instrument zero is a normal piano:

```
F0000017 00000001 00010000 00000000 0F416E79 2053796E
74686573 697A6572 AAE20000 000000AC AAE60000 00130018
0B486172 70736963 686F7264 69616E6F 5069616E 6F000000
00000000 00000000 00000007 00000007 C0010017
```

The $F0000017 says that this is a general event for instrument zero, and the whole thing is $17 (23) long words long. Following that is $15 long words, which actually are a data structure known as a NoteRequest. The last long word $C0010017 says that it is, surprisingly enough, a "note request", and the whole thing was $17 long words long. Why this value is repeated in both places I'm not sure – perhaps there is some sort of integrity check.

Before we get to the code, let's make some Rez definitions so we can create our music – I am currently working on a music editor that will produce these music resources, but I can hardly include the source for that as well in this article (especially since it isn't done yet). It should be done and available in a variety of online places by the time you read this. *[Check out our online places. See p.2 for details – Ed stb]*

'Musi' Resources

I made a simple "Music.r" file (see listing 1) which will allow us to use Rez to create and edit music. Here is our sample input:

```
resource 'Musi' (128) {
  {  /* array header: 1 elements */
     /* [1] */
     0,
     1,
     0x10000,
     kAnyComponentType,
     "Any Synthesizer",
     "Acoustic Grand Piano",
```

```
    1,
    1
  },
  { /* array: 10 elements */
    noteCommand { 0, 37, 64, 1800 },
    restCommand { 1800 },
    noteCommand { 0, 11, 64, 900 },
    restCommand { 900 },
    noteCommand { 0, 28, 64, 1800 },
    noteCommand { 0, 31, 64, 1800 },
    noteCommand { 0, 35, 64, 1800 },
    noteCommand { 0, 4, 64, 1800 },
    restCommand { 3300 },
    markerCommand { 0, 0 }
  }
};
```

This says that we have one instrument, the grand piano, to be played on any synthesizer. If you want to just play around, you can change the last two '1's to other values to play other instruments without having to change the name. I'd advise leaving the polyphony '1' and '0x10000' as is, as well as the synthesizer type and name.

After the header we play a C# in octave 3 (37) at average volume (64) for 3 seconds (1800, since our time scale will have 600 units per second). We rest for those three seconds to let the note play. We then play a B in octave 2 (11) for 1 1/2 second, and also rest to let the note play. We next play a chord of four notes for 3 seconds: E in octave 3 (28), G in octave 3 (31), B in octave 4 (35), and E in octave 1 (4). We let that play and rest for an additional 2 1/2 seconds. We end the thing with our marker command (we could have made that last command part of the rez template like we did for the header, but we might as well mark the end explicitly).

Here's the hex dump of that resource, formatted to show what is going on a little better:

```
00000074 offset to body
6D757369 musi
00000000000000000100000000 flags, etc...
F0000017 instrument 1
   00000001 00010000     Polyphony, etc...
   00000000              Any component
          0F416E792053796E74686573697A6572
          00000000000000000000000000000000
             "Any Synthesizer"
          1441636F7573746963204772616E6420
          5069616E6F00000000000000000000000
             "Acoustic Grand Piano"
   00000001 00000001
C0010017
60000000 End of Commands

Body starts
20960708     Play C# in octave 3
00000708     Rest
202E0384     Play B in octave 2
00000384     Rest
20720708 207E0708 208E0708 20120708 Play Chord
00000CE4     Rest
60000000     End of Commands
```

The Interface

There are many more calls in "QuickTimeComponents.h" than I'll document here, since the focus is to show what it takes to play music in the background of your application. Instead we will just look at the major routines for the tune player component, and ignore both the note allocator and low level music component. This is all from listing 2, my Music.p interface file derived from QuickTimeComponents.h. I'm also going to assume that you've got access to header files and documentation for the Component Manager (found in IM:More Macintosh Toolbox).

```
FUNCTION TuneSetHeader(  tp: TunePlayer;
                         header: Ptr): ComponentResult;
```

This tells the newly created tune player what instruments will be used. We will pass in the header data from our resource.

```
FUNCTION TuneSetTimeScale( tp: TunePlayer;
                           scale: LongInt): ComponentResult;
```

This specifies how many units per second the duration parameter in the music commands stand for. QuickTime uses 600, we use 600. The parameter is actually a TimeScale, but we don't want to have to include all of the QuickTime interface files to find out that it is a longint.

```
FUNCTION TuneGetTimeScale( tp: TunePlayer;
                           VAR scale: LongInt): ComponentResult;
```

This call will return what the current time scale is for the tune player.

```
  FUNCTION TuneQueue( tp: TunePlayer;
                      tune: MusicOpWordPtr;
                      tuneRate: Fixed;
                      tuneStartPosition: LongInt;
                      tuneStopPosition: LongInt;
                      queueFlags: LongInt;
                      callBackProc: ProcPtr;
                      refCon: LongInt): ComponentResult;
```

This is the magic call to actually start playing. You pass in the tune player in tp, and a pointer to the start of the music opwords (make sure that everything is locked down) in tune. TuneRate contains a fixed value which lets you adjust how fast or slow the resulting tune is played. TuneStartPosition and TuneStopPosition specify, in time units, what section of the music to play. The music starts at zero, so to play everything, we pass in 0 and $7FFFFFFF. QueueFlags have the following values:

```
CONST
  kTuneStartNow = 1;
  kTuneDontClipNotes = 2;
  kTuneExcludeEdgeNotes = 4;
  kTuneStartNewMaster = 16384;
```

If no flags are specified, the tune starts playing as soon as any currently playing tune stops (or immediately if no music is currently being played). Up to eight tunes can be queued up at a time.

CallBackProc and refCon are used to help you queue up the next sequence chunk. It would be declared as:

```
PROCEDURE MyTuneCallBackProc(status:TuneStatus; refCon:LongInt);
```

where status is the same as is used in TuneGetStatus, and refCon is whatever you pass into the call of TuneQueue.

```
  TYPE
    TuneStatus = RECORD
      tune, tunePtr: ^LongInt;
      time: longint;
      queueCount, queueSpots: Integer;
      queueTime: LongInt;
      reserved: ARRAY[1..3] OF LongInt;
    END;
  FUNCTION TuneGetStatus (tp: TunePlayer;
    VAR status: TuneStatus): ComponentResult;
```

This routine will give you the status of the currently playing tune. Tune is the current tune, while TunePtr points to where in that tune we currently are. Time is how many time units have passed. QueueCount is how many tunes, including this one are currently queued up. QueueTime is how many time units worth of tunes are queued up waiting to be played.

```
FUNCTION TuneStop(  tp: TunePlayer;
                    stopFlags: LongInt): ComponentResult;
```

This routine is used to stop the specified tune. StopFlags can be one of the following:

```
CONST
  kStopSustain = 1;
  kStopFadeout = 2;
```

This allows you to either let the currently playing note keep playing (which can be annoying), or to let it fade out in a nice manner. If you specify 0, then the music stops abruptly.

```
  FUNCTION TuneResume (tp: TunePlayer): ComponentResult;
```

After you have stopped a tune, this call will let you resume it.

```
  FUNCTION TuneFlush (tp: TunePlayer): ComponentResult;
```

If you decide to not resume a tune, you can call this routine and the next tune queued up should start.

```
  FUNCTION TuneSetVolume (tp: TunePlayer;
    volume: Fixed): ComponentResult;
  FUNCTION TuneGetVolume (tp: TunePlayer): ComponentResult;
```

This pair of routines allows you to change the volume of the playing tune. Don't ask me how the volume is returned in TuneGetVolume, since there is no fixed value returned and no fixed var parameter, but that is what the call is.

```
  FUNCTION TunePreroll (tp: TunePlayer): ComponentResult;
```

This call is important, because it will reserve all the note channels for the instruments, load everything it can into memory, and do any other possible preparation for playing the given music.

```
  FUNCTION TuneUnroll (tp: TunePlayer): ComponentResult;
```

This is the opposite of TunePreroll in that it unreserves all

the note channels that have been locked down. This call is typically called before suspending your application (after stopping the current music), so other applications can play their music.

Putting It All Together

Now that we've got the basic calls, and a simple little bit of music to play, let's look at what calls need to made and in what order. This is the code from a simple application that just gets the music resource and plays it, busy waiting while playing, and quitting when done. We will just comment on the basic calls – to see all the details (where we actually check errors), see listing 3, `MusicTest.p`.

The first thing to do is to load in the resource we are playing and lock it down:

```
h := MusicDescriptionHandle(GetResource('Musi', 128));
HLock(Handle(h));
```

We then call the component manager to have it make a default tune player:

```
tp := OpenDefaultComponent(kTunePlayerType,
                    ResType(LongInt(kAnyComponentType)));
```

The next step is to tell it the time units. We use the value 600 since that is what QuickTime movies normally use.

```
result := TuneSetTimeScale(tp, 600);
```

We then need to tell the tune component what instruments are used. We get this data from our resource.

```
result := TuneSetHeader(tp, @h^^.headerData);
```

Now that we've told it what instruments to use, we want to reserve those instruments and do any other sort of pre-play allocation we can.

```
result := TunePreRoll(tp);
```

We can now say how loud we want to play it. We will use normal volume, which is a fixed point one.

```
result := TuneSetVolume(tp, $10000);
```

Now we make the magic call to actually start the music playing. It will automatically play in the background, there is no extra stuff needed to do (unlike the sound manager). We pass in the start of the body of music, as based on our resource, tell it to play at normal tempo, play everything, start playing now, and we don't have any callbacks.

```
result := TuneQueue(tp, Pointer(ORD4(h^) + h^^.size),
            $10000, 0, $7FFFFFFF, kTuneStartNow, NIL, 0);
```

We can then wait and check to see how we doing. When the music stops, the `queueCount` field of the `TuneStatus` record will drop from 1 to 0. There are other ways to determine if we are done, but we won't get into them, since the `queueCount` field is simple enough.

```
REPEAT
  result := TuneGetStatus(tp, theTuneStatus);
UNTIL Button | (theTuneStatus.queueCount = 0);
```

That's all there is to playing. When we are done, (or in our example, when the user clicks the button), we need to free up everything we allocate.

```
result := TuneStop(tp, kStopFadeout);
err := CloseComponent(tp);
```

That's all there is to playing music. There are many more calls, include ones to present a dialog to the user to allow them to select an instrument, but we don't have space to get into those here. The calls are all found in the QuickTime 2.0 interface files, while the documentation should be in a tech note or future article.

Listing 1 – Music.r

This is our combined header/body resource for playing music with QuickTime 2.0's tune player component.

```
type 'Musi' {
/* First the length of the header & MusicDescription record.  The MusicDescription */
/* record isn't actually used by us or passed as a parameter, but I kept this info */
/* just so it would be easy to convert QuickTime music tracks to 'Musi' resources. */
start: longint = $$CountOf(header) * $17 * 4 + 20 + 4;
  /* next, the media handler type used by QuickTime */
  longint = 'musi';
  longint = 0; /* Reserved 1 */
  integer = 0; /* Reserved 2 */
  integer = 1; /* dataRefIndex */
  longint = 0; /* music flags */
/* and here is the actual start of the music header */
/* we just currently define NoteRequest general events */
  array header {
    bitstring[4] = $F;          /* general event */
    bitstring[12];              /* What instrument */
    bitstring[16] = $0017;      /* length of note request general event */
    /* here is the noteRequest record */
    longint;                    /* Polyphony, usually 1 */
    hex longint;                /* TypicalPolyphony, fixed, usually $10000 */
    longint kAnyComponentType = 0;  /* OSType of synth component */
    pstring[31];    /* Synthesizer name such as "Any Synthesizer" */
    pstring[31];    /* Preferred Instrument name for human use*/
    longint;        /* instrument number if synth-type matches */
    longint;        /* gm number - best matching general MIDI number */
    longint = $C0010017;        /* this was a note request */
  };
  longint = $60000000;          /* The marker at the end of the header */
/* Here is the body of the music */
bodystart:
  array {
    switch {
    case restCommand:
      key bitstring[3] = $0;
      bitstring[5] = 0;         /* unused */
      bitstring[24];            /* duration */
    case noteCommand:
      key bitstring[3] = $1;
      unsigned bitstring[5];    /* instrument */
      unsigned bitstring[6];    /* pitch */
      unsigned bitstring[7];    /* volume */
      unsigned bitstring[11];   /* duration */
    case markerCommand:
      key bitstring[3] = $3;
      bitstring[5] = 0;         /* unused */
      unsigned bitstring[8];    /* subtype */
      bitstring[16];            /* value */
    case controlCommand:
      key bitstring[3] = $2;
      unsigned bitstring[5];    /* instrument */
      unsigned bitstring[8];    /* control number */
      bitstring[16];            /* value */
    };
  };
};
```

Listing 2 – Music.p

```
UNIT Music;
INTERFACE
  USES
    Components;
  CONST
    kMusicComponentType = 'musi';
  TYPE
    MusicDescription = RECORD
        size: LongInt; { including header }
        musicType: LongInt; { 'musi' }
        resvd1: LongInt; { 0 }
        resvd2: Integer;{ 0 }
        dataRefIndex: Integer; { 1 }
        musicFlags: LongInt; { 0 }
        headerData: ARRAY[1..1] OF LongInt;
          { actually, some sort of tone descriptions }
      END;
    MusicDescriptionPtr = ^MusicDescription;
    MusicDescriptionHandle = ^MusicDescriptionPtr;

    TunePlayer = ComponentInstance;
  CONST
    kTuneQueueDepth = 8;
  TYPE
    TuneStatus = RECORD
        tune, tunePtr: ^LongInt;
        time: longint;
        queueCount, queueSpots: Integer;
        queueTime: LongInt;
        reserved: ARRAY[1..3] OF LongInt;
      END;
  CONST
    kStopTuneFade = 1;
    kStopTuneSustain = 2;
    kStopTuneInstant = 4;
    kStopTuneReleaseChannels = 8;
  TYPE
    MusicOpWord = LongInt;
    MusicOpWordPtr = ^MusicOpWord;
  CONST
    kMaxTunePlayerParts = 32;
    tunePlayerRunning = -2100;
    kTunePlayerType = 'tune';

  FUNCTION TuneSetHeader (tp: TunePlayer;
                  header: Ptr): ComponentResult;
  INLINE $2F3C, $4, 4, $7000, $A82A;

  FUNCTION TuneSetTimeScale (tp: TunePlayer;
                  scale: LongInt): ComponentResult;
  INLINE $2F3C, $4, 6, $7000, $A82A;

  FUNCTION TuneGetTimeScale (tp: TunePlayer;
                  VAR scale: LongInt): ComponentResult;
  INLINE $2F3C, $4, 7, $7000, $A82A;

  CONST
    kTuneStartNow = 1;
    kTuneDontClipNotes = 2;
    kTuneExcludeEdgeNotes = 4;
    kTuneStartNewMaster = 16384;

  FUNCTION TuneQueue (tp: TunePlayer;
                  tune: MusicOpWordPtr;
                  tuneRate: Fixed;
                  tuneStartPosition: LongInt;
                  tuneStopPosition: LongInt;
                  queueFlags: LongInt;
                  callBackProc: ProcPtr;
                  refCon: LongInt): ComponentResult;
  INLINE $2F3C, $1C, 10, $7000, $A82A;

  FUNCTION TuneGetStatus (tp: TunePlayer;
                  VAR status: TuneStatus): ComponentResult;
  INLINE $2F3C, $4, 12, $7000, $A82A;
```

```
CONST
  kStopSustain = 1;
  kStopFadeout = 2;

FUNCTION TuneStop (tp: TunePlayer;
                   stopFlags: LongInt): ComponentResult;
INLINE $2F3C, $4, 13, $7000, $A82A;

FUNCTION TuneResume (tp: TunePlayer): ComponentResult;
INLINE $2F3C, 0, 14, $7000, $A82A;

FUNCTION TuneFlush (tp: TunePlayer): ComponentResult;
INLINE $2F3C, 0, 15, $7000, $A82A;

FUNCTION TuneSetVolume (tp: TunePlayer;
                   volume: Fixed): ComponentResult;
INLINE $2F3C, $4, 16, $7000, $A82A;

FUNCTION TuneGetVolume (tp: TunePlayer): ComponentResult;
INLINE $2F3C, 0, 17, $7000, $A82A;

FUNCTION TunePreroll (tp: TunePlayer): ComponentResult;
INLINE $2F3C, 0, 18, $7000, $A82A;

FUNCTION TuneUnroll (tp: TunePlayer): ComponentResult;
INLINE $2F3C, 0, 19, $7000, $A82A;

IMPLEMENTATION
END.
```

Listing 3 – MusicTest.p

```
PROGRAM MusicTest;
  USES
    Components, Music;

  VAR
    h: MusicDescriptionHandle;
    tp: TunePlayer;
    result: ComponentResult;
    theTuneStatus: TuneStatus;
    err: OSErr;

  LABEL
    10; { used for error cleanup }
BEGIN
  { Get the music resource and lock it down }
  h := MusicDescriptionHandle(GetResource('Musi', 128));
  IF h = NIL THEN
    ExitToShell;
  HLock(Handle(h));

  { open the default tune player }
  tp := OpenDefaultComponent(kTunePlayerType,
                    ResType(LongInt(kAnyComponentType)));
  IF tp = NIL THEN
    ExitToShell;

  { tell that we have 600 units per second }
  result := TuneSetTimeScale(tp, 600);
  IF result  noErr THEN
    GOTO 10;

  { Set the header, to tell what instruments are used }
  result := TuneSetHeader(tp, @h^^.headerData);
  IF result  noErr THEN
    GOTO 10;

  { Have it allocate whatever resources are needed }
  result := TunePreRoll(tp);
  IF result  noErr THEN
    GOTO 10;

  { We want to play at normal volume }
  result := TuneSetVolume(tp, $10000);
  IF result  noErr THEN
    GOTO 10;

  { Queue up the music, normal tempo, play everything now }
  result := TuneQueue(tp, Pointer(ORD4(h^) + h^^.size),
                $10000, 0, $7FFFFFFF, kTuneStartNow, NIL, 0);
  IF result  noErr THEN
    GOTO 10;

  REPEAT
    result := TuneGetStatus(tp, theTuneStatus);
    IF result  noErr THEN
      GOTO 10;
  { spin until we click the button or no music left queued up }
  UNTIL Button | (theTuneStatus.queueCount = 0);

  { We get here either by getting an error or having everything finish }
  { Regardless, we need to stop and clean up everything }
10:
  IF result  noErr THEN
    DebugStr('Music result');
  IF tp  NIL THEN
    result := TuneStop(tp, kStopFadeout);
  IF tp  NIL THEN
    err := CloseComponent(tp);

  { And we are done }
END.
```

By R. L. Peskin and S. S. Walther, Landgrove Associates, Flemington, New Jersey

Learning Smalltalk by Examples

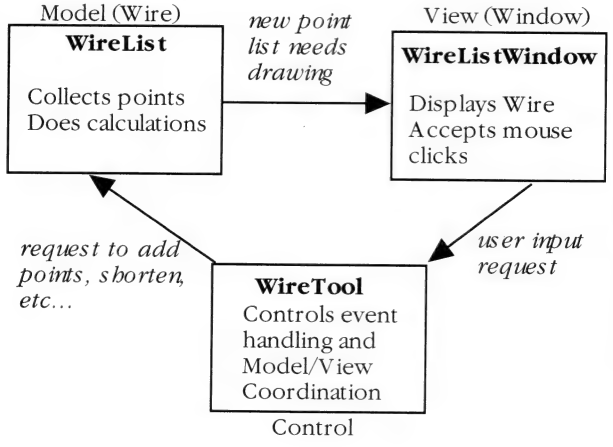
Smalltalk – coming of age and offering an alternative to C and C++

This is the first in a series of articles aimed at those who want to learn something about programming in Smalltalk. Our overall objective is to convey the ease and flexibility of Smalltalk programming to those who may be used to programming in more traditional languages and environments. The approach will be practical; learning by example. We will be using the SmalltalkAgents™ system as the vehicle to illustrate our examples.

The Wire Project

The Wire example is a simple tool to determine the shortest length of wire running between an arbitrarily selected set of points. (If you aren't into wire routing, the same sort of analysis applies to routing a salesperson between cities.) Kent Beck and Ward Cunningham first introduced us to this example at the Tektronix, Inc. Smalltalk classes in 1986. The Wire Project is a good illustration of "tool" building in Smalltalk; that is, integrating the behavior of a Model and the user interface to interact with that Model. The Model refers to your data, perhaps including the computations that generate them. The View is the interface to that data. Commonly this means a visual interface (list, graph, visualization), but it may well include sound, touch, smell, etc. in future platforms. The Controller refers to the set of messages and/or events that facilitate communication and control between the Model and View(s). Our Model is a list of Wire coordinates (list of points), together with the methods (behaviors) that do calculations relevant to those Wire coordinates. Our View is a dynamically changing graphical representation of the Wire points and the lines connecting them. Our Controller is not a formal object, but rather a group of behaviors that effect control; some of these are behaviors of the View and some of the Model.

The Wire tool accepts an arbitrary point from the user's click in a window. The length of the Wire is computed and displayed as points are added. When the user clicks a "Shorten" button, the Wire rearranges its points to minimize the total length and the new arrangement is shown in the window. Here is a diagram of the Wire tool. We'll focus on the implementation of the Model (WireList).

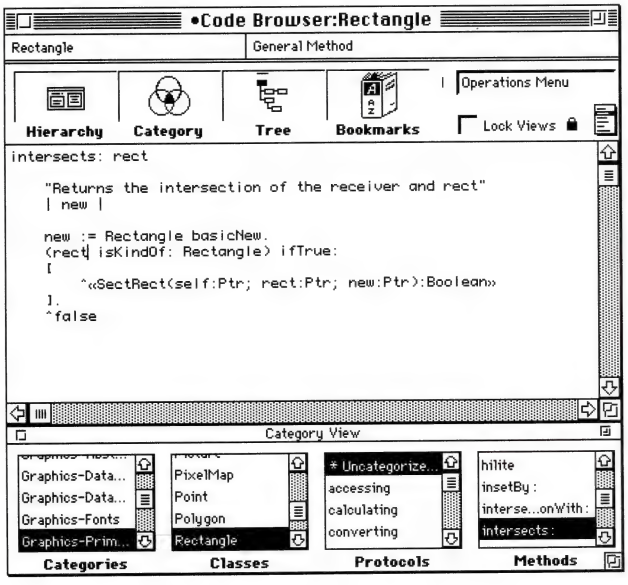


Smalltalk Programming

Smalltalk consist of an "image" which holds objects and their behaviors and other data in the form of interpretable code, and a "virtual machine" which acts as an interpreter of code. (This is an much oversimplified description, but sufficient for present purposes.) Unlike interpreters of old, these virtual machines can run full-out.

The Smalltalk environment is a basic set of tools. It makes Smalltalk a much more productive language to work in than traditional static languages. These tools allow immediate feedback for testing and debugging, as well as incremental development and alteration of code. Changes to individual algorithms (methods) as well as whole classes are easily accomplished interactively. This is in marked contrast to static object-oriented languages such as C++. The Smalltalk environment's tools include classes that are available for the your immediate use. In addition, browsers give you access to the source code for these classes. Debugging tools and text editing fields in browsers and editors let you do code entry, compilation, and execution almost anywhere. Getting from the development process to a standalone application can be done directly within Smalltalk's environment.

The main idea in Smalltalk programming is to program from examples. The primary tool is the Browser, an interactive tool which gives you access to the Smalltalk system code. It also lets you modify code and add your own classes and methods. Here's a Browser view.



A Smalltalk Browser

In the upper left hand corner is the class being browsed (`Rectangle`). The main text edit section shows the method being browsed (`intersects:`). Comments are in quotes. A typical Smalltalk programming statement is: `new := Rectangle basicNew`. The object (in this case the class Rectangle) is sent a message (`basicNew`), and the resulting object assigned to the variable, `new`. Smalltalk syntax refers to the "receiver" (Rectangle in the above example) and the message "selector" (basicNew). Also note the line: `«SectRect(self:Ptr; rect:Ptr; new:Ptr):Boolean»`, this is how a Mac ToolBox call (or other external function) is made in SmalltalkAgents™ (STA). Smalltalk programming is done by editing new or existing methods in the Browser's edit view and saving the edit; saving automatically invokes the compiler. The result is a new or changed method in your "image" with associated byte code (or other binary) representation.

The Category View shown at the bottom has four list views; Categories, Classes, Protocols, and Methods. *Categories* and *Protocols* are for reference and classification purposes only, they have no "programmatic" aspect; Categories group classes together and Protocols group methods. *Classes* list the classes in the Smalltalk class library (including any you have added) and *Methods* list the behaviors (methods) for the selected class.

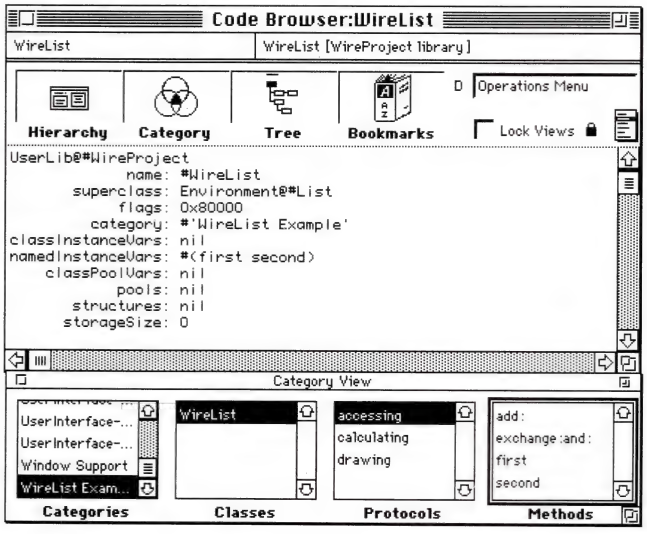
How do we decide what are the objects? How do we decide which classes to subclass? These are common agonies in object-oriented programming. Frankly it may not make that much difference as long as the decisions are reasonable. (For example, a class for our Wires should probably not be a subclass of Rectangle; they have no common behavior.) Some useful rules are 1) If in doubt make your class a subclass of Object (the top level generic class); you can always relocate it later very easily. 2) Choose to subclass under a class (other than Object itself) only if there is a reason to do so. For example, a good reason is so that you can make use of existing methods (behaviors). 3) Don't create a class that doesn't have significant computational responsibilities; classes should have behaviors.

A class has a data structure that contains information (data) about its state. These data are objects called *instance variables.* A class also contains references to behaviors (methods) that let you change or simply read its instance variables. Typically an *object* is a concrete instantiation, that is, member of a class. An object is a specific instance of a class, and has its own set of instance variables. Sometimes classes have behavior and data sufficient to make them do concrete things on their own without instances. Normally a class functions as a template; an object belonging to that class has real values for the class data.

The Wire Model

Our wire is a collection of (geometric) points. One choice is to make the class associated with the Wire a subclass of one of the collection classes. We'll choose to make it a subclass of List. An equally valid (and perhaps better) choice is to make Wire a subclass of Object, and then provide it with an instance variable which itself is a List. In the first case, the Wire will inherit all the behavior of a List; in the second case, the Wire's list instance variable would be accessed when List behavior is needed. For simplicity, we'll chose the former.

We create a subclass of List called "WireList". This operation is usually done by filling in a template in the Browser. Below is the class browser view for WireList. The programmer supplies the name of the subclass as a symbol (#WireList); the superclass (#List), and a list of instance variable names #(first second). (We'll ignore other information such as the paths (e.g. Environment@#List) for now.

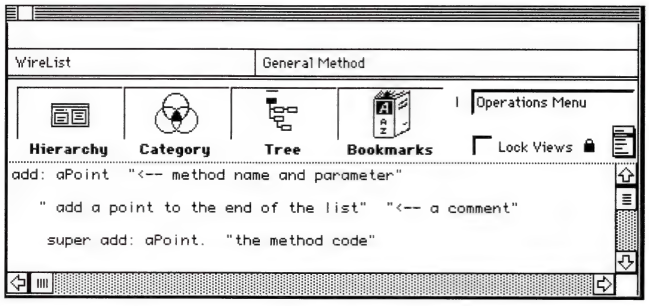


WireList Browser

The category view shows three Protocols, *accessing* (adding point, relocating points, returning specific instance variables), *calculating* (computing wire length, shortening the wire), and *drawing* (supporting calculations for drawing the wire). (Note: You might decide that this last protocol is not appropriate for a List. You might want to create a new subclass for Wire directly under class Object, and have the list as an instance variable. The beauty of Smalltalk is that you can make these "relocations" easily as the program develops, a situation somewhat different than that in C++.)

Now let's fill in the behavior in the first two protocols, starting with *accessing*. First we want to add points to our list. We'll use the following notation: WireList>I>add: to express "WireList class, Instance method, add: aPoint". (An instance method is a behavior for specific objects which are instances of a class; we'll come back to class methods later.)

WireList > I> *add:*



The meaning of *add:* is that the *message* add with a parameter (aPoint) is sent to an object which is an instance WireList. The result is that a new point is added to the specific WireList object. The add: method shown here is sent to "super"; this starts the search for the implementation in WireList's superclass, List, which has the add: method we need. WireList>I>add: is actually not needed, we include it for illustrative purposes. If WireList didn't have a method, add:, the superclass would be tried automatically. If the method is not found there, the search continues until some class in the hierarchy either knew how to do *add:*, or we run out of classes to search and a failure message gets returned. It is instructive to use the Browser tools to see all the different implementations of add: found in the system. (List is an STA class; equivalents in other systems are classes like OrderedCollection.)

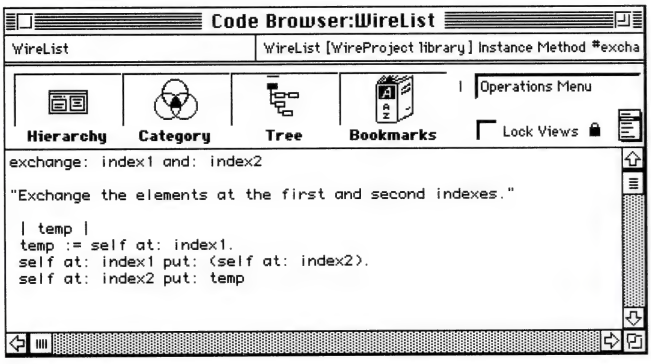
First and *second* are examples of "accessors" – methods to get instance variables. For example,

WireList > I> *second*

This translates to "send the message `position: 1` (go to the first position in the list) to `self`". Self is a reference to the object itself, in this case a specific WireList; self is equivalent to "this" in C++. The semicolon is shorthand for cascading messages. The message "next" to find the next element in the list is sent to the result of the position: 1 message. The result returned, indicated by "^", is the second element in the list.

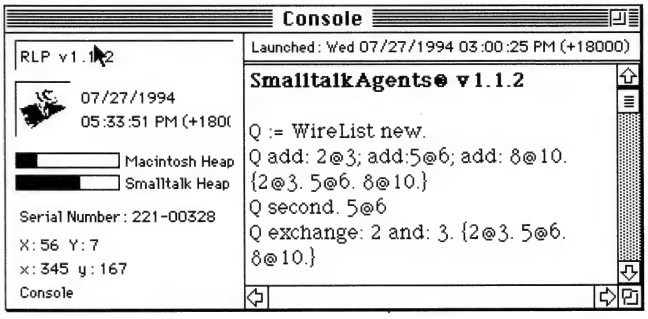
Note that neither "first" nor "second" are actually needed in the Wire tool; they are there for testing purposes. It is good coding form to use accessor methods to get and set instance variables, rather than directly access data structures. The final method in this group does a simple exchange of two list elements.

WireList > I> *exchange:and:*



In this method, *at:put:* is a method of class Object. *At:* is an accessor method of class Object. Temporary variables (whose scope is limited to the method) are denoted by vertical bars, e.g. |temp1 temp2 temp3|. A period is used to denote the end of a statement, and := is the assignment operator. In creating the Wire tool, we figured out that we needed *exchange:and:* during development of one of the algorithms under the *calculating* protocol. Protocol groupings were done as a final step (more on this later).

At this point we can do some computation. We can create a new WireList, put some values in it, access the second element and exchange any two elements. Smalltalk's analog to the "terminal" is a Console or Transcript window. You can also open a window called a "workspace" and do line by line computing. In fact, you can do this in any appropriate code edit field of a Smalltalk window, including the code edit field of the Browser.



A Console Window

We'll show you how some of this interactive stuff works. In the first line of the Console window, we created a new instance (member) of class WireList. *new* is a "constructor" method and is an example of a **class method;** all classes know how to respond to *new* to create new instances. In the second Console line, we added some points, and the next line shows the list of points returned. Then we asked Q for its second member, and 5@6 was returned. Finally we exchanged the 2nd and 3rd elements, and the resulting new list is shown. (Code in a Console can be executed line-by-line or grouped; once a selection is made, you can issue some sort of "execute" or "doIt" command.)

We are now ready to develop the *calculating* protocols, that is, the algorithms to shorten the Wire by seeking the minimum length connecting its points. We'll do this by exchanging point positions in the list until we get a minimum length. Right away, we know we'll need a method to compute distance between points. (This sort of functionality is supplied as one of the methods in class Point in some Smalltalk versions.)

WireList > I> *distance: indx: to: p*

```
distance: indx to: p

    " compute the Euclidean distance between points"

    ^( ((self at: indx) x - p x) squared +
       |((self at: indx) y - p y) squared) squareRoot
```

This method differs from the usual class Point distance method; here we compute distance between a Wire's point referenced via an index and some other point, p. (For simplicity, error checking is not included.) Next we'll look at the method to compute the length of the Wire.

WireList > I> *length*

```
length
 "Answer the length of the wire."
 | total previous |
 total := 0.
 self reset.
 previous :=  self position + 1.
 self
  do:
   [:p |
   total := total + sef distance: previous to: p.
   previous := self indexOf:p.]
 ^total
```

There is a lot in this method. After initializing the length to 0, the List>I>*reset* method is used to set the WireList pointer to the beginning of the list. Smalltalk ordered collections start their indexing at 1, so we set a temporary variable index, 'previous', to the first position. Next we encounter an iteration (do:) over a *block*. Block contexts in Smalltalk are denoted by code contained inside brackets, [ ]. Blocks are a very important part of Smalltalk programming, and have no simple counterpart in C; the context of a Block is preserved even if evaluation is delayed. In the above code, the do: operation iterates over all elements in the list. The syntax [:p | ....code...] denotes that p is a temporary object that will take on different values, the do: assigns p to the points in the list starting at the beginning. Inside the block, we use the method *distance* to compute the total length, which is returned at the end of the iteration.

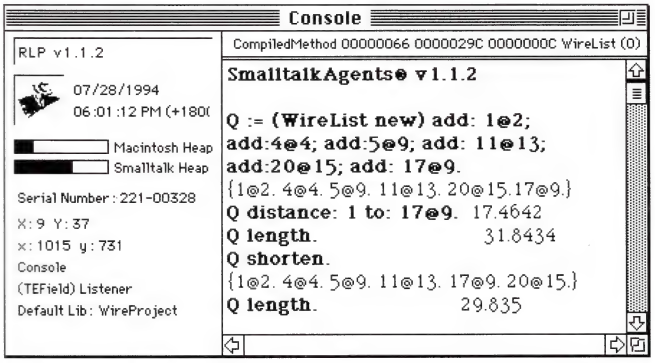
The real action algorithmically in the Wire tool is contained in the method, *shorten*. This is not the appropriate place to discuss the algorithmics employed. It is sufficient that a simplified form of "simulated annealing" is used. We randomly pick integers from the WireList indices and interchange them and check to see if the interchange shortens the Wire.

WireList > I> *shorten*

```
shorten

"Try random changes in the routing order.  Keep
  only changes that shorten the length."

|  minLength i j |

"set the current length to the first guess at the minimum"
 minLength := self length.

"Randomly generate integers and exchange them two at a time. Keep
  the order with the smallest length on each trial"

100 "<-- Arbitrary number guess of trials to reach min. This should
 really be scaled to change with number of elements; e.g.
 might try (self size * 20) instead"
  timesRepeat:
   [i := ((Float random * self size) truncate + 1)asInteger.
   j := ((Float random * self size) truncate + 1)asInteger.
   self exchange: i and: j.
   self length < minLength
    ifTrue: [minLength := self length]
    ifFalse: [self exchange: i and: j]]
```

This method has some new features. 100 timesRepeat: [...] does the code in the outer block 100 times. self size is the size of the Wire list. Float *random* is how random numbers are generated in STA; other systems may use different methods for this. Note the use of *length* and *exchange:and:* that were previously developed. self *length* < minLength returns a Boolean. *ifTrue:[] ifFalse:[]* is a Smalltalk conditional test; if the Boolean is true, one code block is performed, else another code block is performed.

Now we can test out these new methods in a Console window.



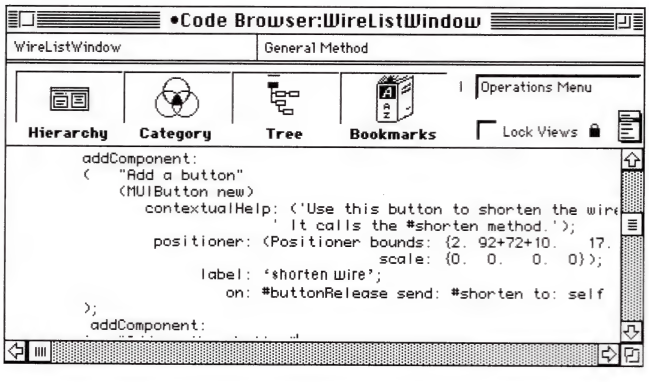
In the first line, a new WireList is created. (Smalltalk code is shown in bold; results are in plain.) Next its distance from the first element to the last point is computed; then the length is calculated. The Wire is told to shorten itself, and a new list results; in this simple case only the last two elements were interchanged. Finally the new length of the shortened wire is obtained. At this point in the development process, we have a fairly complete "model". The next task is to create the tool

per se so that the user can perform these operations on the model via mouse clicks and graphical viewing.

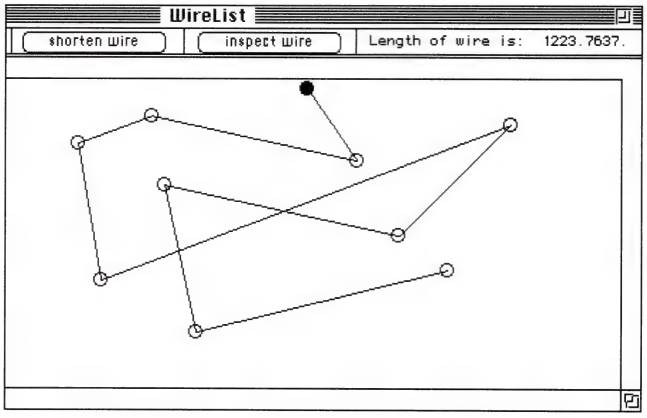
The Wire View and Control

We'll have to leave a complete description of the Wire tool window (WireListWindow) and the control class (WireTool) to the next article, but here's a quick overview. The steps are generally as follows:

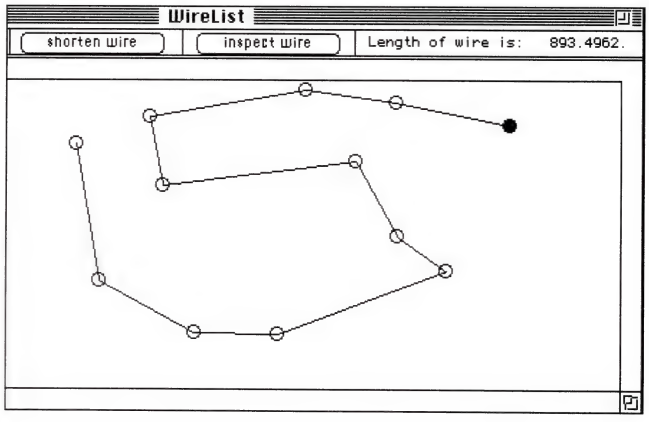
First, we have to make a Window for the tool; in STA we create a subclass of a generic user interface window (UIWindow) which we call WireList Window. Usually this sort of thing is done with a GUI builder facility built into the Smalltalk system. Following is part of the window creation method, namely some code associated with the addition of a <shorten> button. The line '`on: #buttonRelease send: #shorten to: self`' associates the event action (release of the Mouse button) with a method (shorten) that is sent to the WireListWindow when the Mouse is released.



Similar code sets up handling of Mouse action to add points to the Wire. A portion of the WireListWindow is shown below after the user has added some points:



The window looks as follows after pressing the "shorten" button two times (two iterations of the shorten algorithm):



Notice that the wire is shorter and the length has been reduced. How does the user action of clicking at points in the

window result in a) drawing the wire, and b) adding said points to the Wire list? And how do we connect the pressing of the 'shorten' button in the window to invocation of the **WireList > I>** *shorten method?*

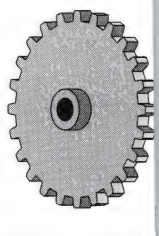
The answer to these questions rests on the fact that, in Smalltalk, views like the above WireListWindow maintain an instance variable (called module in STA) object which is assigned the model (in this case WireList) object instance. The point to be added is supplied by something like "`p := Mouse localPosition`", which returns local window coordinates from the Mouse click. Code like "`module add: p`" adds a point to the list of the selected points. Pressing the 'shorten' button works the same way – the window's `shorten` method simply asks the WireList to shorten; this invokes the wire `shorten` method we described previously.

How the actual drawing of the Wire is accomplished is an interesting point. The details are reasonably complex, and can be Smalltalk system dependent, but the programming is straightforward. When Smalltalk receives an update event, that update request is passed through to the WireListWindow, which handles it by invoking a rendering operation that includes the necessary call to draw the latest Wire based on the current contents of the WireList.

Conclusion

In this article we have tried to present an introduction to Smalltalk programming from a practical point of view. All of the code presented for the WireList can be used (with minor changes) in any Smalltalk version, including public domain ones. If a version of Smalltalk is available, we suggest typing in the WireList code and experimenting. The only way to get a feel for the simplicity and productivity of Smalltalk is to experiment with it. In the next article we will cover the programming support needed for the view and control process, as well as consider some additional tool features such as interactive point relocation and inspection tools.

For SmalltalkAgents™ users, the WireProject code can be found at ftp://qks.com/pub/sta/tutorials/WireList/. A text version of the WireProject suitable for implementation in other Smalltalk versions is available on request (email only) from R. Peskin. Internet: peskin@caip.rutgers.edu, Applelink: D6615, CompuServe: 70372,616. You can also get the project from the usual MacTech Magazine sites or source disk (see p. 2 for details).



By Dave Falkenburg, Apple Computer, Inc.

Sprocket: A Small 7.5-Adept Framework

Introducing the MacTech Magazine tiny application framework!

This month we bring you yet another framework. This one, though, offers a little less than you're probably used to. That's right, less. It's just about as small a framework as you can have and still have some interesting features. This one, written by Apple's Dave Falkenburg, has support for many of the more recent Apple system features, yet comes in well under 33K for a 68K executable.

We plan to use Sprocket as the basis for many of the articles in the coming months. This will allow us to focus on what's being taught (the new code) rather than what you've seen before.

In addition, Dave plans to continue development of Sprocket, so we hope to see more articles detailing how he's making things work, as well as covering new framework features.

We don't have room in this issue to list all of the code, so we're only listing what fits. The complete set of sources will be available on the source code disk and our online sites (see page 2 for info).

Let us hear what you think of this. Do you like the way Sprocket is implemented? Do you have a better way to do something? Is something unclear? Did you find a bug? Is there something you'd like to see added? Let us know; we're aiming to serve your needs – Ed stb

1994 Isn't 1984

With the advent of Apple's System 7.5 release, many of the system extensions developed over the last few years will actually be delivered to the customer in a form they can understand: a single box.

AppleGuide, a faster disk cache, QuickTime 2.0, PC Exchange with Macintosh Easy Open, Macintosh Drag & Drop, and the Thread Manager are some of the enhancements which come along for free with 7.5. Unsuspecting customers also receive a copy of PowerTalk 1.1 and QuickDraw GX 1.01 which they can also install.

It's probably about time to think about adding support for these new features in your own applications. To help you along, here's Sprocket.

Sprocket sports a minimalist design, with just enough to crank out small applications which do their work inside windows. It has plenty of hooks to aid in supporting new Apple technologies, and in many cases has all the necessary support code. It isn't a full-blown class library. In fact, it's only use of C++ objects is to do a very standard Macintosh thing: managing windows.

No! No! No! The Journey is the Journey – The Reward is the Reward.

People not interested in the folklore of how Sprocket came to be can skip ahead to the next section.

Sprocket is the outgrowth of one programmer trying to make sense of all the interesting new things that have been added to Macintosh OS in the past few years. It leverages *heavily* on the ideas presented in recent develop magazine articles by Dean Yu, Steve Falkenburg, and Dave Hershey, as

Dave Falkenburg – Famous for owning furless pets, Dave also has his name in some well-hidden easter eggs in the Macintosh system. He's recently been bludgeoning the 7.5 Process, Layer, and Window managers into submission, and is now pressing on towards the next major release of the system. He makes time at home to dig into the variety of features that his coworkers have been working on, and brings it all together for us here with Sprocket.

well as insights provided by SmartFriends™ all over.

After a dissatisfying experience with C++ and bulky application frameworks, I decided that there was something wrong with creating superfluous objects for things that wouldn't get reused – all it seemed to do was make code less readable to the average programmer, and make things tougher on the compiler.

Like many other folks, I had decided that my application skeleton couldn't be written in C++. Most of my motivation was slow compile times, and the bloated code which came out of CFront at the time. Still wanting flexibility, a decided to use ad hoc "object-oriented C" for my application skeleton. This meant making big structs full of function pointers that get jammed into the refCon of windows, etc. Lots of explicit initialization, and lots of opportunity to type in bugs.

Thankfully, Symantec finally shipped a real C++ compiler and I decided to bite the bullet. C++ was going to be the language I would use.

As it turns out, I just happened to be overhauling my Macintosh skeleton application about the time Dean Yu was working on his first develop article about making floating windows without defiling the Window Manager. As a result, I took an early version of his floating window code and "methodized" it. Similarly, Dean's ideas about smartzooming on comp.sys.mac.programmer were incorporated into my skeleton. After a while, I shelved the project, because I was off to Apple to lend a hand on the PowerMac System Software project.

Maybe it was seeing developers porting their applications to PowerPC, or maybe it was the need to write new code instead of hacking other peoples code, but around the end of the 7.1.2 project I began hacking on my application at home after work. At the time (and until very recently) I called it App. You'll see references to it all through the code (and maybe even here and there in this article).

Around the same time, some folks around here were talking about some Canadian company with PowerPC development tools in their pocket. It turned out that the rumors were true, and that a little 12 person startup had decided to do what two Fortune 500 companies seemingly deemed "impossible".

Metrowerks Code Warrior is an excellent, rapid turnaround environment for building both 68K and PowerPC applications for Macintosh – and it's a hell of a lot cheaper than buying an RS/6000 and trying to weasel an unreleased C compiler from the local Big Blue rep. With new tools in hand, I started adding Mixed Mode Universal ProcPtrs everywhere I needed them, and got it to run on a friend's 6100/60.

About the time I was getting ready to add support for PowerTalk mail and QuickDraw GX, Scott gave me a call and convinced me that this would be a cool thing to share with the rest of the world, especially as a framework for other articles to use as a default basis for what they're teaching.

Dealing with Macintosh Windows – A Few Techniques

Folks familiar with C++ Macintosh programming can skip ahead to the section about the design philosophy employed while writing Sprocket.

A very common technique used by Macintosh programmers is to use the refCon field of the WindowRecord to indicate which routines to call when handling activate, update, mouse clicks, and other events.

One popular technique is to use the windowKind field as an indicator of what routine to call for each possible thing you need to do for a window:

```
void
MyHandleUpdateEvent(WindowPeek theWindowToUpdate)
  {
  GetPort(&oldPort);
  SetPort(theWindowToUpdate);
  BeginUpdate(theWindowToUpdate);

  kindOfWindow = theWindowToUpdate->windowKind;
  switch (kindOfWindow)
    {
    case kDocumentWindow:
      MyDrawingWindowDrawProc();
      break;

    case kDrawingToolsPaletteWindow:
      MyDrawingToolsDrawProc();
      break;

    // ... even more explicit window types follow
    }
  EndUpdate(aWindow);
  SetPort(oldPort);
  }
```

One drawback is the need to maintain a ton of switch statements everywhere in your code. Another less obvious one is that a new type of window cannot be added to the application without recompiling the application. Wouldn't it be nice to let plug-in modules put up their own "first class" windows?

To avoid the problems here, clever programmers have begun to use the refCon field of the WindowRecord to store a table of function pointers. Programs that use this technique include NewsWatcher and CollaboDraw. In this way, all the switch statements can be removed, potentially saving code throughout the file:

```
void
MyHandleUpdateEvent(WindowPtrtheWindowToUpdate)
  {
  GrafPtr  oldPort;
  WindowProcs *procTable =(WindowProcs*)GetWindowRefCon(theWindow);

  GetPort(&oldPort);
  SetPort(theWindowToUpdate);
  BeginUpdate(theWindowToUpdate);
  if (procTable->UpdateProc != NULL)  // call the proc if it
    (*(procTable->DrawProc)());       // isn't null.
  EndUpdate(aWindow);
  SetPort(oldPort);
  }
```

This is flexible, but leads to the need to initialize and maintain these procTables manually, which means a lot of initialization routines like:

```
WindowPtr
MakeMyWindow()
  {
  WindowProcs *myProcTable = NewHandleClear(sizeof(WindowProcs));
  myProcTable->DrawProc = MyDrawingWindowDrawProc;
  // ... and many more things you need to type in.
  }
```

Once you've seen the light, and switch over to using ProcPtrs like this, you've started down the road to object oriented programming in C – object weenies call this stuff "polymorphism". Your event loop only needs to know how to deal with generic windows – no switch statements anywhere. Any new type of window with a different set of WindowProcs can be thought of as a subclass of the base window.

Why Do I Need To Type All This Stuff In?

I don't know about you, but I hate typing. Cut and paste can help here, but it also seems like that's where a lot of bugs get introduced in large programs. Just when you might say: "Gee I can write a tool to automatically generate the initialization code for the window proc tables", you'll notice that there is no need. A C++ compiler can do that for you.

It's simple to create a C++ object whose pointer can be stored in a window's refCon. Other than the "function pointer magic" that the compiler does for you, this method leads to very similar coding as used with the function pointer case:

```
void HandleUpdate(EventRecord * anEvent)
  {
  GrafPtr  oldPort;
  WindowPtr  aWindow = (WindowPtr) anEvent->message;
  TWindow *  wobj;

  GetPort(&oldPort);
  SetPort(aWindow);
  BeginUpdate(aWindow);
  if ((wobj = GetWindowObject(aWindow)) != nil)
    wobj->Draw();
  EndUpdate(aWindow);
  SetPort(oldPort);
  }
```

Besides the desire to dispatch events to windows in a fast and flexible manner, there are other things which "good" Macintosh programs should do. Smart zooming, and patchless floating windows are two of these things.

Both of these things involve overriding the normal behavior of Macintosh windows, and hence are natural for rolling into the the base window methods. If you look at the source, you can see that Dean Yu's advice in **develop** 15 and 17 has been taken to heart. (It's a good thing he wrote those articles before starting to forget about all this Macintosh OS stuff)

Philosophy

The basic approaches used in developing Sprocket were:

- Use C++ objects instead of initializing tables of function pointers. Let the tools do the work – I spent many days debugging stupid "cut and paste" errors in my older code. Now that stable C++ compilers exist for Macintosh, many "nuisance" reasons to avoid C++ are gone. In many environments C++ code compiles just in the same amount of time as C code. There are at least 5 compilers I've used in the last year – MPW C++, Symantec C++, Metrowerks, PPCC, and xlC under AIX on an RS/6000.
- Ignore System 6. If you have a friend with a Macintosh Classic, SE, LC, etc. get them to buy a PowerMac – they'll like it. It isn't that hard to add support back into Sprocket, but it probably isn't worth the added hassles.
- Keep the RAM footprint small for the base functionality so that we don't have to worry about heap fragmentation in the future or possible VM thrashing.
- Fold code snippets used by several subroutines into reusable chunks. Once again – reuse code where it makes sense to keep the code size down. This also tends to improve locality of reference for instruction cache accesses.
- Keep use of C++ objects down to only areas where there will typically be a lot of shared code or reuse. MacApp's Nothing application used to be 300K! We don't have a TApplication – there's usually only ONE application anyway, why carry around base-class code you might not use just because C++ insists on creating references to it in the __vtbls?
- Sprocket is not a class library full of everything you ever had to do in CS101: If you want one of those, go buy it – it'll have code written by genuine computer scientists that know more about algorithms than the average programmer. Lots of folks like the Booch class libraries.
- Use the Thread Manager in lieu of lots of IdleProcs. We still have Idle handlers for windows, but they really aren't needed if the application can rely on using the Thread Manager.

A Quick Look At The Sources

The source directory is broken down into three folders:

:Interfaces:	Definitions for Sprocket-supplied objects and utility functions.
:Lib:	Implementation of Sprocket's core routines and objects.
:AppSpecific:	Example code for building your own application using Sprocket.

As a rule of thumb, you shouldn't need to change things inside :Interfaces: or :Lib:, while :AppSpecific: contains code designed for tweaking/replacement.

The code has been successfully built using Metrowerks CodeWarrior 4 and tested on a Macintosh IIsi. It is in the process of being run through even more compilers – I had some trouble using PPCC, so there could be something wrong with the code *[we'll be grateful for any corrections you readers send in – Ed stb].*

The coding conventions are based on MacApp and the unofficial C++ style guide published in an early develop article. I haven't gone crazy with `const` parameters and the like, because like most of us, I'm still learning things about Bjarne's creation. The indented braces are a habit of mine, since the Process Manager (and Finder) were typed this way.

Now on to the files themselves – first the core Sprocket sources:

`:Interfaces:AppConditionals.h`

Conditional compilation flags kept in a header. MPW and Unix folk tend to keep these things in Makefiles, but that really means that you have to use make. In this manner, we can potentially build Sprocket in both Unix/MPW and THINK/MW environments. If you do source code control, it's also handy to keep conditionals in a source file.

`:Interfaces:StandardMenus.h`

Constants defining some standard MENU IDs. It would be nice to rip out menu IDs and item numbers from application code to abstract menu choices from the position in the menu bar. Just about every other framework does. We may do that eventually.

`:Interfaces:AppLib.h`

Function prototypes and declarations for all the utility functions and useful global variables provided in the less object oriented parts of Sprocket.

`:Lib:AppLib.cp`

`main()` lives in here. The heart of the Macintosh application. Routines include toolbox initialization, the main event loop, and cleanup routines. Points of interest include the use of dynamic run and sleep values inside the main event loop.

`:Interfaces:AppleEventHandling.h`
`:Lib:AppleEventHandling.cp`

Code for handling the required Apple event suite (kAEOpenApplication, kAEOpenDocuments, kAEPrintDocuments, and kAEQuitApplication). Support for opening AOCE letters is also supported. Once we add OSA scripting support, expect this file to grow by leaps and bounds.

`:Interfaces:DialogUtils.h`
`:Lib:DialogUtils.cp`

Useful utility functions for using modal dialogs and alerts in the System 7 world. Contains an auto-resized error alert mechanism and lots of FilterProcs needed to prevent update events from blocking background applications. For more information about why we do all these things see the Macintosh Technote: Pending Update Perils. Support for calling the Thread Manager `YieldToAnyThread` inside the filterProcs is also provided.

:Lib:Preferences.cp

Routines for dealing opening and/or creating a preferences file. *[Am I the only one who thought that develop article on preferences files was a bit of overkill? – drf]*

:Interfaces:Window.h
:Lib:Window.cp

These are especially interesting, and are listed at the end of the article for your reading enjoyment.

Our favorite C++ object (so far). TWindow is the base class for defining a Macintosh window. Every window created directly by Sprocket is built using this object. When a Macintosh window is created, the pointer to a corresponding C++ object is stashed in the refCon.

I've taken care to make creating windows from application code as simple as:

```
TWhizbangWindow * myWindow = new
TWhizbangWindow(<WhizzyParameters>);
```

Creating new kinds of windows is as easy as overriding the `MakeNewWindow` method, and providing unique drawing & event handling methods.

Right now, both floating and normal windows are supported. Modal windows (e.g., windows that come above floaters and cause them to be deactivated) are next on the list for implementation.

Weird C++ ALERT: This isn't as easy as it looks because C++ has a habit of not letting you call virtual methods in an inherited class from within a constructor. From within the bottom-most derived window class, we call back to an inherited method, `CreateWindow`, to do common window creation, instead of creating the window from within `TWindow::TWindow()`.

:Interfaces:DialogWindow.h
:Lib:DialogWindow.cp

A class called TDialogWindow which allows for the trivial creation of Dialog Manager-based windows. It's kinda skanky right now because it temporarily patches `FrontWindow` in order to fool IsDialogEvent and DialogSelect into working with Dean's floater technique.

:Interfaces:SplashWindow.h
:Lib:SplashWindow.cp

Object-based implementation of the splash screen. Mostly gratuitous, but a fun example of subclassing TWindow.

Application Specific Stuff

:AppSpecific:App.cp

Recipe code for the application-specific things. It implements the following required functions, as well as an about box dialog:

```
OSErr    SetupApplication(void);
OSErr    TearDownApplication(void);
void     HandleMenu(TWindow * topWindowObj,long menuCode);
void     ConvertClipboard(void);
OSErr    OpenNewDocument(void);
OSErr    OpenDocument(LetterDescriptor *,void *);
OSErr    PrintDocument(LetterDescriptor *,void *);
Boolean  QuitApplication(void);
```

:AppSpecific:DocWindow.cp
:AppSpecific:DocWindow.h

A window which spins arrows using threads and draws a grow box.

:AppSpecific:PreferencesDialogWindow.cp
:AppSpecific:PreferencesDialogWindow.h

A totally bogus example showing how to override TDialogWindow. Preferences dialogs that look like this are less than nice to the novice user. Cool applications derive default preferences from the user's use of the application, not from clever imitations of the System 6 Control Panel.

:AppSpecific:ToolWindow.cp
:AppSpecific:ToolWindow.h

An empty floating tool palette. Eventually it'd be nice to define tool palette template resources, but for now we use the parameter as a WIND resource id.

Nothing against my brother or anything, but I just can't read the code in CollaboDraw without getting hives. Here's my first attempt at building some C++ window classes for dealing with AOCE:

:Unfinished AOCE Stuff:MailableWindow.h
:Unfinished AOCE Stuff:MailableWindow.cp

A base class for an AOCE mailable window. Intended to be placed within :Lib: once all the mailer commands and Undo support have been added to Sprocket.

:Unfinished AOCE Stuff:MailableDocWindow.cp
:Unfinished AOCE Stuff:MailableDocWindow.h

A simple subclass of TMailableWindow intended to be added to :AppSpecific: when it is more complete.

Have fun reading through the code. There should be a couple of even less obvious tips and tricks in there that I've since forgotten about that might be useful. Like I said, Sprocket has picked up a lot of ideas from other sample code through the years.

Speaking of other code, remember that Sprocket is not the be-all, end-all Macintosh framework. It's really a minimalist shell that makes it easy to demonstrate cool things in the magazine or building little hacks.

Other Frameworks

There are a ton of other Macintosh specific and/or cross platform frameworks that can be of great help. We'll be focusing on Sprocket for articles here in the magazine, but these other frameworks have a lot to offer for full-fledged applications. A number of them are advertised right here in the magazine. Here are a few more (in no particular order):

MacApp – The first framework for building applications supplied by Apple. It had it's beginnings as Object Pascal framework, but has been converted C++ over the last few years, and can now be used to build Power Macintosh applications. MacApp 3.x started integrating System 7 functionality and began to embrace C++.

TCL – A THINK C Objects based framework. Since Symantec didn't have "real C++" until a few years ago, the most popular compiler used by Mac developers was essentially locked out of MacApp. TCL is still a popular framework, and the 2.0 version has been ported to the PowerPC.

AppsToGo – Eric Soldan of Apple DTS has a C-based application framework. This framework is heavily built upon being able to dynamically instantiate user interface elements from templates. Because it sprang forth out of the DTS.Lib effort in the early 90's, lots of utility functions for doing "useful things" in MacOS are also included. Some things built using AppsToGo include ClickBook and Kibitz, as well as a bunch of other commercial applications. You can find it at ftp://ftp.apple.com/dts/mac/sc/apps.to.go

QuickApp – Another MacApp-flavored application framework. This is probably a lot like Sprocket, but is probably better tested, and leverages off more experiences. Apps can be as small as 50K. $149, e-mail emergent@aol.com.

MetroWerks PowerPlant – A C++ class library for building Macintosh Applications. There is a lot of very cool stuff inside PowerPlant, including support for floating windows, resource-based user interfaces, robust exception handling, AppleEvent Object model support, and lots of very useful classes for using the Thread Manager. Uses multiple inheritance to get more code reuse. Also has it's own coding conventions similar to the MacApp-style, but also containing hints for whether or not the object "pulls in lots of other stuff", or it's a lightweight, reusable "L"ibrary class. Greg Dow, the original author of TCL, is the principal force behind PowerPlant.

Visual C++, MFC – Microsoft's cross platform development strategy – don't write Mac software, just write Windows code. Probably a good choice for people porting vertical market applications from Windows to Mac on a very tight schedule.

OpenDoc – A whole new way of building applications. Stop thinking about applications – parts is parts. The idea here is that you essentially write a C++ object that can be embedded inside any document. It is important to note that OpenDoc isn't really a framework in the sense of providing default "do the right thing" methods for parts. Instead it provides the class hierarchy alone. A cross-platform parts framework (OPF, the OpenDoc Parts Framework, descended from BedRock) is also under development.

OLE – Microsoft's component software strategy. Instead of eliminating the application altogether, this is kind of a super-plug in that lets Microsoft use your code within their Office Suite, and other OLE-enabled applications.

Taligent – Some really cool C++ stuff, but few details have

yet been disclosed in a public forum.

WHAT TO EXPECT IN THE FUTURE:

- Making Sprocket more robust with a clean, ARM-inspired exception model.
- More support for AppleScript, and hence more studly AppleGuide abilities.
- Adding QuickDraw GX-aware Printing, and possibly GX window classes.
- Changes to transparently adapt to new stuff coming from Apple.
- And, of course, cool demos and new articles based on Sprocket.

And now to a few selected listings. We'll show Window.h, Window.cp, and AppLib.h. They're especially interesting, and they filled up all of our available space for this month. To see the rest of the files, check out this month's source disk or the online sites (see p. 2 for details).

Window.h

/* Contains: Definition of TWindow, a base class which provides a framework for building way-cool windows which even John Sullivan would be happy with. Floating windows and "smart zooming" algorithms are based on code samples provided by Dean Yu. Written by: Dave Falkenburg, Dean Yu
Copyright: © 1993-94 by Dave Falkenburg, all rights reserved. */

```
#ifndef     _WINDOW_
#define     _WINDOW_

#ifndef     __TYPES__
#include <Types.h>
#endif

#ifndef     __WINDOWS__
#include <Windows.h>
#endif
#ifndef     __EVENTS__
#include <Events.h>
#endif
#ifndef     __DRAG__
#include <Drag.h>
#endif

typedef  short  WindowTemplateID;

class  TWindow
  {
public:
  enum WindowType
    {
    kNormalWindow = 0,
    kFloatingWindow,
    kModalWindow
    };
            TWindow();
  virtual        ~TWindow();

  // Event routing methods
  virtual  Boolean    EventFilter(EventRecord * theEvent);
```

Methods you shouldn't need to override, but might need to

```
  virtual void    CreateWindow(WindowType typeOfWindowToCreate
                                  = kNormalWindow);
  virtual  void   Select(void);
  virtual  void   Drag(Point startPoint);
  virtual void    Nudge( short horizontalDistance,
                         short verticalDistance);
  virtual void    Grow(Point startPoint);
  virtual void    Zoom(short zoomState);
  virtual  void   ShowHide(Boolean showFlag);
```

Methods which MUST be overridden:

```
  virtual WindowPtr MakeNewWindow(WindowPtr behindWindow) = 0;
```

Methods which probably should be overridden

```
  virtual  void   AdjustCursor(EventRecord * anEvent);
  virtual void    Idle(EventRecord * anEvent);
  virtual void    Activate(Boolean activating);
  virtual void    Draw(void);
  virtual void    Click(EventRecord * anEvent);
  virtual  void   KeyDown(EventRecord * anEvent);
  virtual void    GetPerfectWindowSize(Rect * perfectSize);
  virtual  void   GetWindowSizeLimits(Rect * limits);
  virtual void    AdjustForNewWindowSize(  Rect * oldRect,
                                           Rect * newRect);
  // Window property accessor methods...
  //     ...watch for new ones when we add scripting support

  virtual  Boolean    IsVisible(void);
  virtual  Boolean    CanClose(void);
  virtual  Boolean    Close(void);
  virtual  Boolean    DeleteAfterClose(void);
  virtual  void       DoEditMenu(short item);

  // Methods for use with the Drag Manager

  virtual  OSErr  HandleDrag(DragTrackingMessage dragMessage,
                             DragReference theDrag);
  virtual  OSErr    DragEnterWindow(DragReference theDrag);
  virtual  OSErr    DragInWindow(DragReference theDrag);
  virtual  OSErr    DragLeaveWindow(DragReference theDrag);
  virtual  OSErr    HandleDrop(DragReference theDragRef);

protected:
  WindowPtr       fWindow;
  WindowType      fWindowType;

  Boolean         fIsVisible;
  };
```

Utility Functions:

```
// Don't you just wish you didn't have to do this?
```

```
pascal WindowPtr  GetNewColorOrBlackAndWhiteWindow(
          short windowID, void *wStorage, WindowPtr behind);
pascal WindowPtr  NewColorOrBlackAndWhiteWindow(
          void *wStorage, const Rect *boundsRect,
          ConstStr255Param title, Boolean visible,
          short theProc, WindowPtr behind,
          Boolean goAwayFlag, long refCon);

TWindow *       GetWindowObject(WindowPtr aWindow);
WindowPtr       FrontModalWindow(void);
WindowPtr       LastModalWindow(void);
WindowPtr       FrontFloatingWindow(void);
WindowPtr       LastFloatingWindow(void);
WindowPtr       FrontNonFloatingWindow(void);
void        HiliteAndActivateWindow( WindowPtr aWindow,
                                     Boolean active);
void        SuspendResumeWindows(Boolean resuming);
void        HiliteWindowsForModalDialog(Boolean hiliting);

pascal OSErr    CallWindowDragTrackingHandler(
                    DragTrackingMessage message,
                    WindowPtr theWindow,
                    void *handlerRefCon,
                    DragReference theDragRef);
pascal OSErr    CallWindowDragReceiveHandler(
                    WindowPtr theWindow,
                    void *handlerRefCon,
                    DragReference theDragRef);
#endif
```

Window.cp

/* Contains: Implementation of TWindow, a base class which provides a framework for building way-cool windows which even John Sullivan would be happy with. Floating windows and "smart zooming" algorithms are based on code samples provided by Dean Yu. Tim Craycroft, the guy making the window manager do all this work for you has also been a great help.

Written by: Dave Falkenburg
Copyright: © 1993-94 by Dave Falkenburg, all rights reserved.

To Do:
Make sure invisible windows can be created & managed
Handle modal windows as another class of windows
Fix activate bugs when showing and hiding windows
Window positioning methods (getters and setters)
Display Manager support
Changes to support AEObject model
*/

```
#include <Types.h>
#include <Windows.h>
#include <Errors.h>
#include <Script.h>     // for GetMBarHeight()
#include <LowMem.h>     // for LMGetWindowList()

#include "AppLib.h"
#include "Window.h"

const short     kFloatingWindowKind    = 1000;
const short     kNormalWindowKind    = 1001;
const WindowPtr   kNoFloatingWindows     = (WindowPtr) -1;
const short     kScreenEdgeSlop      = 4;
const short     kSpaceForFinderIcons= 64;
const short     kMinimumTitleBarHeight= 21;
const short     kMinimumWindowSize     = 32;

static void     HiliteShowHideFloatingWindows(
                          Boolean hiliting,Boolean hiding);

static void     FindScreenRectWithLargestPartOfWindow(
          WindowPtr aWindow, Rect *theBestScreenRect,
          GDHandle * theBestDevice);
static pascal void  CalculateWindowAreaOnDevice(
          short depth, short deviceFlags, GDHandle targetDevice,
          long userData);

struct CalcWindowAreaDeviceLoopUserData
  {
  GDHandle fScreenWithLargestPartOfWindow;
  long   fLargestArea;
  Rect   fWindowBounds;
  };
```



```
TWindow::TWindow()
  {
  }
TWindow::~TWindow()
  {
  }
```

TWindow::CreateWindow

```
void TWindow::CreateWindow( WindowType typeOfWindowToCreate
/* = kNormalWindow */)
  {
  WindowPtr  behindWindow,oldFrontMostWindow;

  if (typeOfWindowToCreate == kModalWindow)
    {
    DebugStr("\pModal windows aren't supported yet");
    fWindowType = kFloatingWindow;
    return;
    }
  else if (typeOfWindowToCreate == kFloatingWindow)
    {
    behindWindow = (WindowPtr) -1;
    oldFrontMostWindow = FrontWindow();

    fWindowType = kFloatingWindow;
    }
  else if (typeOfWindowToCreate == kNormalWindow)
    {
    behindWindow = LastFloatingWindow();

    fWindowType = kNormalWindow;

    if (behindWindow == kNoFloatingWindows)
      oldFrontMostWindow = nil;
    else
      oldFrontMostWindow = (WindowPtr)
                  ((WindowPeek) behindWindow)->nextWindow;
    }
```

```
    fWindow = this->MakeNewWindow(behindWindow);
    fIsVisible = ((WindowPeek) fWindow)->visible;

    if (fWindow)
      {
      SetWRefCon(fWindow,(long) this);

      if (typeOfWindowToCreate == kModalWindow)
        {
        DebugStr("\pCan't create Modal windows yet");
        }
      else if (typeOfWindowToCreate == kFloatingWindow)
        {
        ((WindowPeek) fWindow)->windowKind = kFloatingWindowKind;

        // make sure the other window stays hilited
        if (oldFrontMostWindow)
          HiliteAndActivateWindow(oldFrontMostWindow,true);
        }
      else if (typeOfWindowToCreate == kNormalWindow)
        {
        ((WindowPeek) fWindow)->windowKind = kNormalWindowKind;

        // unhighlight the old front window
        if (oldFrontMostWindow)
          HiliteAndActivateWindow(oldFrontMostWindow,false);

        // hilite the new window...
        HiliteAndActivateWindow(fWindow,true);
        }
      }
    }
```

TWindow::AdjustCursor

```
void
TWindow::AdjustCursor(EventRecord * /* anEvent */)
    {
    }
```

TWindow::Idle

```
void
TWindow::Idle(EventRecord * /* anEvent */)
    {
    }
```

TWindow::Activate

```
void
TWindow::Activate(Boolean /* activating */)
    {
    }
```

TWindow::Draw

```
void
TWindow::Draw(void)
    {
    }
```

TWindow::Click

```
void
TWindow::Click(EventRecord * /* anEvent */)
    {
    }
```

TWindow::KeyDown

```
void
TWindow::KeyDown(EventRecord * /* anEvent */)
    {
    }
```

TWindow::Select

```
void
TWindow::Select(void)
    {
    WindowPtr   currentFrontWindow;

    if (fWindowType == kFloatingWindow)
      currentFrontWindow = FrontWindow();
    else if (fWindowType == kNormalWindow)
      currentFrontWindow = FrontNonFloatingWindow();
    else
      {
      }

    if (currentFrontWindow != fWindow)
      {
      if (fWindowType == kFloatingWindow)
        BringToFront(fWindow);
      else
        {
        WindowPtr   lastFloater = LastFloatingWindow();

    // If there are no floating windows, just call SelectWindow like the good ol' days
        if (lastFloater == kNoFloatingWindows)
          SelectWindow(fWindow);
        else
          {
          // Deactivate the window currently in front.
          HiliteAndActivateWindow(currentFrontWindow,false);

// Bring it behind the last floating window and activate it.  Note that Inside Mac 1
// states that you need to call PaintOne() and CalcVis() on a window if you are using
// SendBehind() to bring it closer to the front.  With Sys 7, this is no longer necessary.
          SendBehind(fWindow,lastFloater);
          HiliteAndActivateWindow(fWindow,true);
          }
        }
      }
    }
```

TWindow::Drag

```
void
TWindow::Drag(Point startPoint)
    {
    GrafPtr     savePort;
    KeyMap    theKeyMap;
    Boolean     commandKeyDown = false;
    RgnHandle   draggingRegion;
    long    dragResult;
    WindowPeek windowAsWindowPeek = (WindowPeek) fWindow;

    if (WaitMouseUp())      // de-bounce?
      {
      // Set up the Window Manager port.
      GetPort(&savePort);
      SetPort(gWindowManagerPort);
      SetClip(GetGrayRgn());

      // Check to see if the command key is down.
      GetKeys(theKeyMap);
      commandKeyDown = ((theKeyMap[1] & 0x8000) != 0);

      if (commandKeyDown)
        {
        // We're not going to change window ordering, so make sure that we don't
        // drag in front of other windows which may be in front of ours.
        ClipAbove(windowAsWindowPeek);
        }
      else if (fWindowType != kFloatingWindow)
        {
        // We're dragging a normal window, so make sure that we don't drag in
        // front of any floating windows.
        ClipAbove((WindowPeek) FrontNonFloatingWindow());
        }

      // Drag an outline of the window around the desktop.
      // NOTE: DragGrayRgn destroys the region passed in, so make a copy
      draggingRegion = NewRgn();
      CopyRgn(windowAsWindowPeek->strucRgn,draggingRegion);
      dragResult = DragGrayRgn(draggingRegion, startPoint,
        &gDeskRectangle, &gDeskRectangle, noConstraint, nil);
      DisposeRgn(draggingRegion);
      SetPort(savePort);    // Get back to old port

      if ((dragResult != 0) && (dragResult != 0x80008000))
        {
        this->Nudge((dragResult & 0xFFFF),(dragResult >> 16));
        }
      }
    if (!commandKeyDown)
      Select();
    }
```

TWindow::Nudge

```
void
TWindow::Nudge(short horizontalDistance, short
verticalDistance)
    {
    WindowPeek windowAsWindowPeek = (WindowPeek) fWindow;
    short    newHorizontalPosition,newVerticalPosition;

    newHorizontalPosition = (short) (**windowAsWindowPeek
      ->contRgn).rgnBBox.left + horizontalDistance;
```

```
  newVerticalPosition = (short) (**windowAsWindowPeek
    ->contRgn).rgnBBox.top + verticalDistance;
  MoveWindow(fWindow,newHorizontalPosition,
             newVerticalPosition, false);
  }
```

TWindow::Grow

```
void
TWindow::Grow(Point startPoint)
  {
  GrafPtr  oldPort;
  long newSize;
  Rect oldWindowRect,resizeLimits;

  GetPort(&oldPort);
  GetWindowSizeLimits(&resizeLimits);
  newSize = GrowWindow(fWindow,startPoint,&resizeLimits);
  if (newSize)
    {
    oldWindowRect = fWindow->portRect;
    SizeWindow(fWindow,(short) newSize,
               (short) (newSize >> 16),true);
    SetPort(fWindow);
    this->AdjustForNewWindowSize(&oldWindowRect,
                                 &fWindow->portRect);
    }
  SetPort(oldPort);
  }
```

TWindow::Zoom

```
void
TWindow::Zoom(short zoomState)
  {
  GrafPtr  oldPort;
  FontInfo systemFontInfo;
  short    titleBarHeight;
  Rect     bestScreenRect,perfectWindowRect,scratchRect;
  short    amountOffscreen;
  WindowPeek windowAsWindowPeek = (WindowPeek) fWindow;
  GDHandle bestDevice;

  GetPort(&oldPort);

  // Figure out the height of the title bar so we can properly position
  // a window. The algorithm is stolen from the System 7.x 'WDEF' (0)
  // This probably isn't the best thing to do: A better way might be
  // to diff the structure and content region rectangles?
  SetPort(gWindowManagerPort);
  GetFontInfo(&systemFontInfo);
  titleBarHeight = (short) (systemFontInfo.ascent +
                            systemFontInfo.descent + 4);
  if ((titleBarHeight % 2) == 1)
    titleBarHeight--;
  if (titleBarHeight < kMinimumTitleBarHeight)
    titleBarHeight = kMinimumTitleBarHeight;

  // Only do the voodoo magic if we are really "zooming" the window.
  if (zoomState == inZoomOut)
    {
    FindScreenRectWithLargestPartOfWindow(
          fWindow,&bestScreenRect,&bestDevice);
    bestScreenRect.top += titleBarHeight;

    this->GetPerfectWindowSize(&perfectWindowRect);
    OffsetRect(&perfectWindowRect,-perfectWindowRect.left,
               -perfectWindowRect.top);

    // Take the zero-pined perfect window size and move it to
    // the top left of the  window's content region.
    OffsetRect(&perfectWindowRect,
               (**windowAsWindowPeek->contRgn).rgnBBox.left,
               (**windowAsWindowPeek->contRgn).rgnBBox.top);

    // Does perfectWindowRect fit completely on the best screen?
    SectRect(&perfectWindowRect,&bestScreenRect,&scratchRect);
    if (!EqualRect(&perfectWindowRect, &scratchRect))
      {
      // SectRect sez perfectWindowRect doesn't completely fit on the screen,
      // so bump the window so that more of it fits.
      // Make sure that the left edge of perfectWindowRect is forced onto the best
      // screen. This is in case we are bumping the window to the right.
      amountOffscreen = bestScreenRect.left
                        - perfectWindowRect.left;
      if (amountOffscreen > 0)
        {
        perfectWindowRect.left += amountOffscreen;
        perfectWindowRect.right += amountOffscreen;
        }

      // Make sure that the left edge of perfectWindowRect is forced
      // onto the best screen. This is in case we are bumping
      // the window downward to a new screen.
      amountOffscreen = bestScreenRect.top
                        - perfectWindowRect.top;
      if (amountOffscreen > 0)
        {
        perfectWindowRect.top += amountOffscreen;
        perfectWindowRect.bottom += amountOffscreen;
        }

      // If right edge of window falls off the screen,
      //     Move window to the left until the right edge IS on the screen
      //     OR the left edge is at bestScreenRect.left
      amountOffscreen = perfectWindowRect.right
                        - bestScreenRect.right;
      if (amountOffscreen > 0)
        {
        // Are we going to push the left edge offscreen? If so, change the
        // offset so we move the window all the way over to the left.
        if ((perfectWindowRect.left - amountOffscreen)
                < bestScreenRect.left)
          amountOffscreen = perfectWindowRect.left
                            - bestScreenRect.left;
        perfectWindowRect.left -= amountOffscreen;
        perfectWindowRect.right -= amountOffscreen;
        }

      // If bottom edge of window falls off the screen,
      //     Move window to up until the bottom edge IS on the screen
      //     OR the top edge is at bestScreenRect.top
      amountOffscreen = perfectWindowRect.bottom
                        - bestScreenRect.bottom;
      if (amountOffscreen > 0)
        {
        // Are we going to push the top edge offscreen? If so, change the
        // offset so we move the window just to the top.
        if ((perfectWindowRect.top - amountOffscreen)
                < bestScreenRect.top)
          amountOffscreen = perfectWindowRect.top
                            - bestScreenRect.top;
        perfectWindowRect.top -= amountOffscreen;
        perfectWindowRect.bottom -= amountOffscreen;
        }
      SectRect(&perfectWindowRect, &bestScreenRect,
               &scratchRect);
      if (!EqualRect(&perfectWindowRect, &scratchRect))
        {
        // The edges of the window still fall offscreen,
        // so make the window smaller until it fits.
        if (perfectWindowRect.bottom > bestScreenRect.bottom)
          perfectWindowRect.bottom = bestScreenRect.bottom;

        // If the right edge is still falling off,
        //     save space for Finder's disk icons as well.
        if (perfectWindowRect.right > bestScreenRect.right)
          {
          perfectWindowRect.right = bestScreenRect.right;

          // If we were on the main screen, leave room for Finder icons, too.
          if (bestDevice == GetMainDevice())
            perfectWindowRect.right -= kSpaceForFinderIcons;
          }
        }
      }
    // Stash our new rectangle inside of the Window's dataHandle
    // so that ZoomWindow does the right thing.
    (**((WStateDataHandle)
      (windowAsWindowPeek->dataHandle))).stdState
      = perfectWindowRect;
    }
  // Don't forget to set the port to the window being zoomed
  // Why, you ask? Because IM-IV-50 says to; otherwise you die
  SetPort(fWindow);

  Rect oldWindowRect = fWindow->portRect;
```

```
    ZoomWindow(fWindow,zoomState,false);
    this->AdjustForNewWindowSize(&oldWindowRect,
                                 &fWindow->portRect);
    SetPort(oldPort);
    }
```

TWindow::ShowHide

```
void
TWindow::ShowHide(Boolean showFlag)
    {
    // Here we need the "::" in front of ShowHide to indicate we are calling
    // the global function, and not the method ShowHide. Unintended recursion
    // can do bad things to the unsuspecting programmer.
    // Some C++ programmers would always prepend the "::" on function calls.
    ::ShowHide(fWindow,showFlag);
    fIsVisible = showFlag;
    }
```

TWindow::EventFilter

```
Boolean
TWindow::EventFilter(EventRecord * /* theEvent */)
    {
    return false;
    }
```

TWindow::GetPerfectWindowSize

```
void
TWindow::GetPerfectWindowSize(Rect *perfectSize)
    {
    *perfectSize = qd.screenBits.bounds;
    }
```

TWindow::GetWindowSizeLimits

```
void
TWindow::GetWindowSizeLimits(Rect *limits)
    {
    limits->top = limits->left = kMinimumWindowSize;
    limits->right = gDeskRectangle.right - gDeskRectangle.left;
    limits->bottom = gDeskRectangle.bottom
                   - gDeskRectangle.top;
    }
```

TWindow::AdjustForNewWindowSize

```
void
TWindow::AdjustForNewWindowSize(  Rect * /* oldRect */,
                                  Rect * /* newSize */)
    {
    }
```

TWindow::IsVisible

```
Boolean
TWindow::IsVisible(void)
    {
    return fIsVisible;
    }
```

TWindow::CanClose

```
Boolean
TWindow::CanClose(void)
    {
    return true;
    }
```

TWindow::Close

```
Boolean
TWindow::Close(void)
    {
    WindowPtr  newFrontWindow = nil;

    if (FrontNonFloatingWindow() == fWindow)
      newFrontWindow = (WindowPtr) ((WindowPeek) fWindow)
                                    ->nextWindow;
    DisposeWindow(fWindow);
    if (newFrontWindow)
      HiliteAndActivateWindow(newFrontWindow,true);
    return true;
    }
```

TWindow::DeleteAfterClose

```
Boolean
TWindow::DeleteAfterClose(void)
    {
    return true;
    }
```

TWindow::DoEditMenu

```
void
TWindow::DoEditMenu(short /* menuCode */)
    {
    }
```

TWindow::HandleDrag

```
OSErr
TWindow::HandleDrag(DragTrackingMessage dragMessage,
                    DragReference theDrag)
    {
    OSErr  result = dragNotAcceptedErr;
    switch (dragMessage)
      {
      case dragTrackingEnterWindow:
        result = this->DragEnterWindow(theDrag);
        break;
      case dragTrackingInWindow:
        result = this->DragInWindow(theDrag);
        break;
      case dragTrackingLeaveWindow:
        result = this->DragLeaveWindow(theDrag);
        break;
      default:
        break;
      }
    return result;
    }
```

TWindow::DragEnterWindow

```
OSErr
TWindow::DragEnterWindow(DragReference /* theDrag */)
    {
    return dragNotAcceptedErr;
    }
```

TWindow::DragInWindow

```
OSErr
TWindow::DragInWindow(DragReference /* theDrag */)
    {
    return dragNotAcceptedErr;
    }
```

TWindow::DragLeaveWindow

```
OSErr
TWindow::DragLeaveWindow(DragReference /* theDrag */)
    {
    return dragNotAcceptedErr;
    }
```

TWindow::HandleDrop

```
OSErr
TWindow::HandleDrop(DragReference /* theDrag */)
    {
    return dragNotAcceptedErr;
    }
```

Utility Functions used for floating windows

GetWindowObject

```
TWindow *
GetWindowObject(WindowPtr aWindow)
    {
    short  wKind;

    if (aWindow != nil)
      {
      wKind = ((WindowPeek) aWindow)->windowKind;
      if (wKind >= userKind)
        {
        // All windowKinds >= userKind are based upon TWindow
        return (TWindow *) GetWRefCon(aWindow);
        }
      }
    return (TWindow *) nil;
    }
```

Utility functions

GetNewColorOrBlackAndWhiteWindow

```
pascal WindowPtr
GetNewColorOrBlackAndWhiteWindow(short windowID,
                                 void *wStorage, WindowPtr behind)
    {
    if (gHasColorQuickdraw)
      return GetNewCWindow(windowID,wStorage,behind);
    else
      return GetNewWindow(windowID,wStorage,behind);
    }
```

NewColorOrBlackAndWhiteWindow

```
pascal WindowPtr
NewColorOrBlackAndWhiteWindow( void *wStorage,
      const Rect *boundsRect, ConstStr255Param title,
      Boolean visible, short theProc, WindowPtr behind,
      Boolean goAwayFlag, long refCon)
    {
```

```
  if (gHasColorQuickdraw)
    return NewCWindow(wStorage, boundsRect, title, visible,
                      theProc, behind, goAwayFlag, refCon);
  else
    return NewWindow(wStorage, boundsRect, title, visible,
                     theProc, behind, goAwayFlag,refCon);
  }
```

LastFloatingWindow

```
WindowPtr
LastFloatingWindow(void)
  {
  WindowPeek aWindow = (WindowPeek) FrontWindow();
  WindowPtr  lastFloater = (WindowPtr) kNoFloatingWindows;

  while (aWindow && (aWindow->windowKind
         == kFloatingWindowKind))
    {
    if (aWindow->visible)
      lastFloater = (WindowPtr) aWindow;

    aWindow = (WindowPeek) aWindow->nextWindow;
    }
  return(lastFloater);
  }
```

FrontNonFloatingWindow

```
WindowPtr
FrontNonFloatingWindow(void)
  {
  WindowPeek aWindow = (WindowPeek) LMGetWindowList();

  // Skip over floating windows
  while (aWindow && (aWindow->windowKind
         == kFloatingWindowKind))
    aWindow = (WindowPeek) aWindow->nextWindow;

  // Skip over invisible, but otherwise normal windows
  while (aWindow && (aWindow->visible == 0))
    aWindow = (WindowPeek) aWindow->nextWindow;

  return (WindowPtr) aWindow;
  }
```

HiliteAndActivateWindow

```
void
HiliteAndActivateWindow(WindowPtr aWindow,Boolean active)
  {
  GrafPtr    oldPort;
  TWindow  * wobj = GetWindowObject(aWindow);

  if (aWindow)
    {
    HiliteWindow(aWindow,active);
    if (wobj != nil)
      {
      GetPort(&oldPort);
      SetPort(aWindow);
      wobj->Activate(active);
      SetPort(oldPort);
      }
    }
  }
```

SuspendResumeWindows

```
void
SuspendResumeWindows(Boolean resuming)
  {
  // When we suspend/resume, hide/show all the visible floaters
  HiliteShowHideFloatingWindows(resuming,true);
  }
```

HiliteWindowsForModalDialog

```
void
HiliteWindowsForModalDialog(Boolean hiliting)
  {
  // When we display a modal dialog, we need to unhighlight
  // all visible floaters. We also need to re-hilite them afterwards.
  HiliteShowHideFloatingWindows(hiliting,false);
  }
```

HiliteShowHideFloatingWindows

```
void
HiliteShowHideFloatingWindows(Boolean hiliting,Boolean
dohiding)
  {
  WindowPeek aWindow;
  TWindow *  wobj;

  HiliteAndActivateWindow(FrontNonFloatingWindow(),hiliting);
  aWindow = LMGetWindowList();
  while (aWindow && aWindow->windowKind
    == kFloatingWindowKind)
    {
    wobj = GetWindowObject((WindowPtr) aWindow);

    // If we are hiding or showing, only hide/show windows that were
    // visible to begin with.
    // NOTE:  We use our copy of the visible flag so we can
    //        automatically show floaters on a resume event.
    // NOTE:  Since this isn't a method of TWindow, we don't really need the "::"
    //        on ShowHide, but as long as we're trying to avoid ambiguity...
    if (dohiding && (wobj != nil) && (wobj->IsVisible()))
      ::ShowHide((WindowPtr) aWindow,hiliting);

    // All floaters are hilited if any floater is hilited
    HiliteWindow((WindowPtr) aWindow,hiliting);
    aWindow = (WindowPeek) aWindow->nextWindow;
    }
  }
```

Routines used for dealing with windows and multiple screens

CalculateWindowAreaOnDevice

```
pascal void
CalculateWindowAreaOnDevice(short /* depth */,
      short /* deviceFlags */,GDHandle targetDevice,long userData)
  {
  CalcWindowAreaDeviceLoopUserData *  deviceLoopDataPtr;
  long                windowAreaOnThisScreen;
  Rect                windowRectOnThisScreen;

  deviceLoopDataPtr = (CalcWindowAreaDeviceLoopUserData *)
                      userData;

  SectRect(&deviceLoopDataPtr->fWindowBounds,
           &(**targetDevice).gdRect,&windowRectOnThisScreen);
  OffsetRect(&windowRectOnThisScreen,
             -windowRectOnThisScreen.left,
             -windowRectOnThisScreen.top);
  windowAreaOnThisScreen = windowRectOnThisScreen.right
                           * windowRectOnThisScreen.bottom;
  if (windowAreaOnThisScreen > deviceLoopDataPtr->fLargestArea)
    {
    deviceLoopDataPtr->fLargestArea = windowAreaOnThisScreen;
    deviceLoopDataPtr->fScreenWithLargestPartOfWindow
        = targetDevice;
    }
  }

DeviceLoopDrawingUPP CallCalcWindowAreaOnDevice =
    NewDeviceLoopDrawingProc(&CalculateWindowAreaOnDevice);
```

FindScreenRectWithLargestPartOfWindow

```
void
FindScreenRectWithLargestPartOfWindow(WindowPtr aWindow,
         Rect *theBestScreenRect,GDHandle * theBestDevice)
  {
  RgnHandle               copyOfWindowStrucRgn;
  CalcWindowAreaDeviceLoopUserData  deviceLoopData;

  // Use DeviceLoop to find out what GDevice contains the largest
  // portion of the supplied window.
  // NOTE:  Assumes thePort == the Window Manager Port because we using
  //        the window strucRgn, not contRgn.
  deviceLoopData.fScreenWithLargestPartOfWindow = nil;
  deviceLoopData.fLargestArea = -1;
  deviceLoopData.fWindowBounds = (**(((WindowPeek) aWindow)
                                  ->contRgn)).rgnBBox;
  copyOfWindowStrucRgn = NewRgn();
  CopyRgn(((WindowPeek) aWindow)->strucRgn,copyOfWindowStrucRgn);

  DeviceLoop(copyOfWindowStrucRgn,
             CallCalcWindowAreaOnDevice,
             (long) &deviceLoopData,singleDevices);

  DisposeRgn(copyOfWindowStrucRgn);

  *theBestDevice =
      deviceLoopData.fScreenWithLargestPartOfWindow;
  *theBestScreenRect =
```

```
    (**(deviceLoopData.fScreenWithLargestPartOfWindow)).gdRect;

  // Leave some space around the edges of the screen so window look good, AND
  // if the best device is the main screen, leave space for the Menubar
  InsetRect(theBestScreenRect,kScreenEdgeSlop,kScreenEdgeSlop);
  if (GetMainDevice()
        == deviceLoopData.fScreenWithLargestPartOfWindow)
    theBestScreenRect->top += GetMBarHeight();
  }
```

Drag Manager callback routines which dispatch to a window's method

CallWindowDragTrackingHandler

```
pascal OSErr
CallWindowDragTrackingHandler(DragTrackingMessage dragMessage,
  WindowPtr theWindow,void * /* refCon */,DragReference theDrag)
  {
  TWindow *wobj = GetWindowObject(theWindow);

  if (wobj)
    return(wobj->HandleDrag(dragMessage,theDrag));
  else
    return dragNotAcceptedErr;
  }
```

CallWindowDragReceiveHandler

```
pascal OSErr
CallWindowDragReceiveHandler(WindowPtr theWindow,
                  void * /* refCon */,DragReference theDrag)
  {
  TWindow *wobj = GetWindowObject(theWindow);

  if (wobj)
    return(wobj->HandleDrop(theDrag));
  else
    return dragNotAcceptedErr;
  }
```

AppLib.h

/* Contains: Prototypes for the "guts" of a Macintosh application.

Copyright: © 1993 by Dave Falkenburg, all rights reserved. */

```
#ifndef    _APPLIB_
#define    _APPLIB_

#include "AppConditionals.h"
#include "Preferences.h"
#include <Types.h>
#include <Windows.h>
#include <Dialogs.h>
#include <Menus.h>
#include <Files.h>
#include <AppleEvents.h>
#include <StandardFile.h>
#include <OCEStandardMail.h>
#include "Window.h"

#if qUseQuickDrawGX
#include <FixMath.h>// make sure we don't use GX lame #define of "fixed1"
#include <graphics types.h>
#endif
```

useful macros

```
#if qDebug
#define  DebugMessage(x) DebugStr(x)
#else
#define  DebugMessage(x)
#endif
```

Resource IDs

```
#define  kErrorAlertID                    128
#define  kStandardCloseAlertID            129
#define  kStandardCloseWithNewPubsAlertID 130
#define  kCoreErrorStrings                128
#define  kUnsupportedSystemSoftware     1
#define  kNeedsThreadManager            2
#define  kStandardCloseStrings            129
#define  kQuittingStr                   1
#define  kClosingStr                    2
#define  kPreferencesFileStrings          130
#define  kPreferencesFileName           1
#define  kSplashPictureID                 128

#if    qUseQuickDrawGX
// When using GX, we want to create a 300K graphics heap
// NOTE: I have no idea what to use as a number here!
#define  kGraphicsHeapSize    (300 * 1024)
#endif
```

Useful functions provided by App:

```
void     HandleEvent(EventRecord *anEvent);
void     HandleClose(WindowPtr aWindow);
short    StandardAlert(short alertID, short defaultItem=ok,
                       short cancelItem              = 0,
                       ModalFilterUPP customFilterProc= nil);
void     ErrorAlert(short stringList,short whichString);
void     FatalErrorAlert(short stringList,short whichString);

extern ModalFilterUPP  StandardDialogFilter;
extern ModalFilterYDUPP  StandardDialogFilterYD;
extern void     PseudoClickInDialogItem(DialogPtr theDialog,
                                        short itemToClick);
enum StandardCloseResult
  {
  kSaveDocument = 1,
  kCancelSaveDocument = 2,
  kDontSaveDocument = 3
  };

StandardCloseResult StandardCloseDocument(
  const StringPtr documentType,StringPtr documentName,
  Boolean hasNewEditions, Boolean quitting);

OSErr  CheckAppleEventForMissingParams(
  AppleEvent *theAppleEvent);

short  OpenPreferencesResFile(void);

// AOCE "FrontWindow"-equivalent routine for the Standard Mail package
extern FrontWindowUPP  FrontWindowProcForAOCEUPP;
```

Globals

```
extern Boolean        gDone;
extern Boolean        gMenuBarNeedsUpdate;
```

```
extern Boolean          gHasColorQuickdraw;
extern Boolean          gHasThreadManager;
extern Boolean          gHasDragManager;
extern Boolean          gHasAOCE;
extern Boolean          gHasDisplayManager;

#if qInlineInputAware
extern Boolean          gHasTextServices;
extern Boolean          gHasTSMTE;
#endif

#if qUseQuickDrawGX
extern Boolean          gHasQuickDrawGX;
extern long             gQuickDrawGXVersion;
extern long             gQuickDrawGXPrintingVersion;
extern gxGraphicsClient  gQuickDrawGXClient;
#endif

extern GrafPtr          gWindowManagerPort;
extern Rect             gDeskRectangle;
extern RgnHandle        gMouseRegion;

extern short            gPreferencesRsrcRefNum;
```

Routines that the application MUST supply:

```
extern OSErr  SetupApplication(void);
extern void TearDownApplication(void);
extern void HandleMenu(TWindow * topWindowObj,long menuCode);
extern void ConvertClipboard(void);

extern OSErr  OpenNewDocument(void);
extern OSErr  OpenDocument(LetterDescriptor *,void *);
extern OSErr  PrintDocument(LetterDescriptor *,void *);
extern Boolean  QuitApplication(void);

#endif
```

AppLib.cp

```
/* Contains: The "guts" of a Macintosh application. Written by Dave and many other
SmartFriends™ Copyright: © 1993-94 by Dave Falkenburg, all rights reserved. */
#ifdef SystemSevenOrLater
#undef SystemSevenOrLater
#endif
#define  SystemSevenOrLater  1

#include <limits.h>    // For LONG_MAX

#include <Types.h>
#include <Quickdraw.h>
#include <Fonts.h>
#include <Menus.h>
#include <Windows.h>
#include <Dialogs.h>
#include <Desk.h>
#include <Events.h>
#include <AppleEvents.h>
#include <DiskInit.h>

#if qUseETO15Interfaces
#include <Gestalt.h>
#include <CodeFragments.h>
#include <Devices.h>
#else
#include <GestaltEqu.h>
#include <FragLoad.h>
#endif

#include <ToolUtils.h>
#include <Traps.h>
#include <LowMem.h>

#include <Threads.h>
#include <Drag.h>
#include <Editions.h>
#include <OCEStandardMail.h>

#if qInlineInputAware
#include <TextServices.h>
#include <TSMTE.h>
#endif
```



```
#include "AppLib.h"
#include "StandardMenus.h"
#include "Window.h"
#include "SplashWindow.h"
#include "MailableWindow.h"
#include "AppleEventHandling.h"

#if qUseQuickDrawGX
#include <graphics macintosh.h>
#include <graphics routines.h>
#include <PrintingManager.h>
#endif
```

Function Prototypes

```
void main(void);
void MainEventLoop(void);
void HandleMouseDown(TWindow * topWindowObj, EventRecord * anEvent);
void HandleUpdate(EventRecord * anEvent);
void HandleClose(WindowPtr aWindow);
```

Globals

```
Boolean         gDone = false;
Boolean         gMenuBarNeedsUpdate = true;

Boolean         gHasColorQuickdraw = false;
Boolean         gHasThreadManager = false;
Boolean         gHasDragManager = false;
Boolean         gHasAppleScript = false;
Boolean         gHasAOCE = false;
Boolean         gHasDisplayManager = false;

GrafPtr         gWindowManagerPort;
Rect            gDeskRectangle;
RgnHandle       gMouseRegion = nil;

short           gPreferencesRsrcRefNum;

#if qInlineInputAware
Boolean         gHasTextServices = false;
```

```
Boolean         gHasTSMTE = false;
#endif

#if  qUseQuickDrawGX
Boolean         gHasQuickDrawGX = false;
long            gQuickDrawGXVersion = 0;
long            gQuickDrawGXPrintingVersion = 0;
gxGraphicsClient  gQuickDrawGXClient;

// PrintingEventOverride is our generic event handler for QuickDrawGX.
// It alows us to handle events while the QuickDrawGX movable modal
// printing dialogs are displayed.
// ¿ Really should move to a GX-specific place ?
```

PrintingEventOverrideForGX

```
OSErr
PrintingEventOverrideForGX(EventRecord *anEvent,
                           Boolean filterEvent)
  {
  if (!filterEvent)
    switch (anEvent->what)
      {
      case mouseDown:
      case keyDown:
      case autoKey:
        break;
      default:
        HandleEvent(anEvent);
      }
  return noErr;
  }
#endif

// Values that can be adjusted by other application code to change
// the behavior of the MainEventLoop.
//
// Rules of thumb:
//     Increase gXXXRunQuantum (and decrease gXXXSleepQuantum) when:
//         The application has many threads running that need time
//     Decrease gXXXRunQuantum when:
//         Sending AppleEvents to other applications
//         Launching other applications
//         Running in the background

unsigned long gForegroundRunQuantum = 0;
unsigned long gForegroundSleepQuantum = GetCaretTime();
unsigned long gBackgroundRunQuantum = 0;
unsigned long gBackgroundSleepQuantum = LONG_MAX;

// Globals used to "tune" the performance of MainEventLoop
// (assume we'll be starting in the foreground)

static unsigned long gRunQuantum = gForegroundRunQuantum;
static unsigned long gSleepQuantum = gForegroundSleepQuantum;

#ifdef powerc
#ifndef  __MWERKS__
QDGlobals  qd;
#endif
#endif
```

main

```
void
main(void)
  {
  long   feature;

  MaxApplZone();
  MoreMasters(); MoreMasters(); MoreMasters(); MoreMasters();

  InitGraf(&qd.thePort);
  InitFonts();
  InitWindows();
  InitMenus();
  TEInit();
  InitDialogs(nil);

  if (GetToolTrapAddress(_Unimplemented)
      == GetOSTrapAddress(_Gestalt))
    FatalErrorAlert(kCoreErrorStrings,
                    kUnsupportedSystemSoftware);

  if (Gestalt(gestaltQuickdrawFeatures,&feature) == noErr)
    gHasColorQuickdraw = ((feature & (1 << gestaltHasColor)) != 0);

  TSplashWindow * splashWindow = new TSplashWindow;

  if ((Gestalt(gestaltAppleEventsAttr,&feature) == noErr)
      && (feature & (1 << gestaltAppleEventsPresent)))
    {
    // Figure out if we need to do AppleEvent recording
    gHasAppleScript = (feature & (1 << gestaltScriptingSupport));
    }
  else
    FatalErrorAlert(kCoreErrorStrings,
                    kUnsupportedSystemSoftware);

#if  qInlineInputAware
  if ((Gestalt(gestaltTSMgrVersion,&feature) == noErr)
      && (feature >= 1))
    {
    gHasTextServices = true;
    if (Gestalt(gestaltTSMTEAttr, &feature) == noErr)
      gHasTSMTE = (feature & (1 << gestaltTSMTEPresent));
    }
#endif

  if (Gestalt(gestaltThreadMgrAttr,&feature) == noErr)
    {
#ifdef powerc
    // If running on a PowerPC, make sure that we not only have the 68K Thread
    // Manager, but also the PowerPC shared library, too.  Because of the wonders
    // of weak linking and out of memory errors we need to also check to make
    // sure that an entrypoint in the library is there, too.
    if ((Ptr) NewThread != kUnresolvedSymbolAddress)
      gHasThreadManager = ((feature
    & ((1 << gestaltThreadMgrPresent)
      | (1 << gestaltThreadsLibraryPresent))) != 0);
#else
    gHasThreadManager = ((feature & (1 <<
```

```
gestaltThreadMgrPresent)) != 0);
#endif
    }

  // Check for and install Drag Manager callbacks
  if (Gestalt(gestaltDragMgrAttr,&feature) == noErr)
    {
#ifdef powerc
    // If running on a PowerPC, make sure that we not only have the
    // 68K Drag Manager, but also the PowerPC shared library, too.
    if ((Ptr) NewDrag != kUnresolvedSymbolAddress)
      gHasDragManager = ((feature
        & ((1 << gestaltDragMgrPresent)
         | (1 << gestaltPPCDragLibPresent))) != 0);
#else
    gHasDragManager = ((feature & (1<<gestaltDragMgrPresent))!=0);
#endif

    if (gHasDragManager)
      {
      InstallTrackingHandler( NewDragTrackingHandlerProc
                              (CallWindowDragTrackingHandler),
                              (WindowPtr) nil,nil);
      InstallReceiveHandler( NewDragReceiveHandlerProc
                              (CallWindowDragReceiveHandler),
                              (WindowPtr) nil,nil);
      }
    }
  // Check for Display Manager
  if (Gestalt(gestaltDisplayMgrAttr,&feature) == noErr)
    gHasDisplayManager
      = ((feature & (1 << gestaltDisplayMgrPresent)) != 0);

  // Check for and initialize AOCE Standard Mail package if it exists
  if ((Gestalt(gestaltSMPMailerVersion,&feature) == noErr)
      && (feature != 0))
    {
#ifdef powerc
    if ((Ptr) SMPInitMailer != kUnresolvedSymbolAddress)
      gHasAOCE = (SMPInitMailer(kSMPVersion) == noErr);
#else
    gHasAOCE = (SMPInitMailer(kSMPVersion) == noErr);
#endif
    }

#if qUseQuickDrawGX
  // Check for and initialize QuickDrawGX
  if (Gestalt(gestaltGXVersion, &gQuickDrawGXVersion)==noErr)
    if (Gestalt(gestaltGXPrintingMgrVersion,
                &gQuickDrawGXPrintingVersion) == noErr)
#ifdef powerc
      if ((Ptr) GXEnterGraphics != kUnresolvedSymbolAddress)
        gHasQuickDrawGX = true;
#else
      gHasQuickDrawGX = true;
#endif

  if (gHasQuickDrawGX)
    {
//  gQuickDrawGXClient = GXNewGraphicsClient(nil, kGraphicsHeapSize,
//                                           (gxClientAttribute) 0);
    GXEnterGraphics();
    GXInitPrinting();
    }
#endif

  InstallAppleEventHandlers(); // Install our AppleEvent Handlers

  // Setup desktop rectangle for dragging windows around
  GetWMgrPort(&gWindowManagerPort);
  gDeskRectangle = (**GetGrayRgn()).rgnBBox;

  // Get the default menubar
  SetMenuBar(GetNewMBar(rMenuBar));
  AddResMenu(GetMHandle(mApple),'DRVR');

  gPreferencesRsrcRefNum = OpenPreferencesResFile();

  if (SetupApplication() == noErr)
    {
    delete splashWindow; // get rid of the splash screen
```



```
    MainEventLoop();
    TearDownApplication();
    }

#if qUseQuickDrawGX
  if (gHasQuickDrawGX)   // Tear down QuickDrawGX
    {
    GXExitPrinting();
//  GXDisposeGraphicsClient(gQuickDrawGXClient);// Dies a horrible death for me?
    GXExitGraphics();
    }
#endif
  }
```

MainEventLoop

```
void
MainEventLoop(void)
  {
  EventRecord  anEvent;
  unsigned long nextTimeToCheckForEvents = 0;

  while (!gDone)
    {
    if (gMenuBarNeedsUpdate)
      {
      gMenuBarNeedsUpdate = false;
      DrawMenuBar();
      }
    if ((gRunQuantum == 0) ||
      (TickCount() > nextTimeToCheckForEvents))
      {
      nextTimeToCheckForEvents = TickCount() + gRunQuantum;
      (void) WaitNextEvent(everyEvent,&anEvent,
                           gSleepQuantum,gMouseRegion);
      HandleEvent(&anEvent);
      }
    if (gHasThreadManager)
      YieldToAnyThread();
```

```
    }
  }
```

HandleEvent

```
void
HandleEvent(EventRecord *anEvent)
  {
  TWindow  * wobj;

  if (anEvent->what != updateEvt)
    wobj = GetWindowObject(FrontNonFloatingWindow());
  else
    wobj = GetWindowObject((WindowPtr) anEvent->message);
  if (wobj != nil)
    wobj->AdjustCursor(anEvent);
  if ((wobj != nil) && wobj->EventFilter(anEvent))
    return;
  else switch (anEvent->what)
    {
    case nullEvent:
      if (wobj != nil)
        wobj->Idle(anEvent);
      break;
    case mouseDown:
      HandleMouseDown(wobj,anEvent);
      break;
    case keyDown:
    case autoKey:
      if (anEvent->modifiers & cmdKey)
        HandleMenu(wobj,MenuKey((short) anEvent->message
                                & charCodeMask));
      else if (wobj != nil)
        wobj->KeyDown(anEvent);
      break;
    case updateEvt:
      HandleUpdate(anEvent);
      break;
    case diskEvt:
      if (anEvent->message >> 16)
        {
        static Point  where = {50,50};
        (void) DIBadMount(where,anEvent->message);
        }
      break;
    case osEvt:
      switch ((anEvent->message & osEvtMessageMask) >> 24)
        {
        case mouseMovedMessage:
          break;
        case suspendResumeMessage:
          if (anEvent->message & resumeFlag)
            {
            gRunQuantum = gForegroundRunQuantum;
            gSleepQuantum = gForegroundSleepQuantum;
            }
          else
            {
            gRunQuantum = gBackgroundRunQuantum;
            gSleepQuantum = gBackgroundSleepQuantum;
            }
          SuspendResumeWindows(
                  (anEvent->message & resumeFlag) != 0);
          if (anEvent->message & convertClipboardFlag)
            ConvertClipboard();
          break;
        }
      break;
    case kHighLevelEvent:
      (void) AEProcessAppleEvent(anEvent);
      break;
    default:
      break;
    }
  }
```

HandleMouseDown

```
void
HandleMouseDown(TWindow * topWindowObj,EventRecord *anEvent)
  {
  WindowPtr  aWindow;
  short    partCode;
  TWindow    *wobj;

  partCode = FindWindow(anEvent->where,&aWindow);
  wobj = GetWindowObject(aWindow);
  switch(partCode)
    {
    case inMenuBar:
      HandleMenu(topWindowObj,MenuSelect(anEvent->where));
      break;
    case inSysWindow:
      SystemClick(anEvent,aWindow);
      break;
    case inContent:
      if (wobj)
        {
        GrafPtr  oldPort;

        GetPort(&oldPort);
        SetPort(aWindow);
        GlobalToLocal(&anEvent->where);
        wobj->Click(anEvent);
        SetPort(aWindow);
        }
      break;
    case inDrag:
      if (wobj)
        wobj->Drag(anEvent->where);
      break;
    case inGrow:
      if (wobj)
        wobj->Grow(anEvent->where);
      break;
    case inGoAway:
      if (TrackGoAway(aWindow,anEvent->where))
        HandleClose(aWindow);
      break;
    case inZoomIn:
    case inZoomOut:
      if (TrackBox(aWindow,anEvent->where,partCode) && (wobj))
        wobj->Zoom(partCode);
      break;
    default:
      break;
    }
  }
```

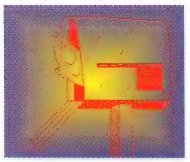
HandleUpdate

```
void HandleUpdate(EventRecord * anEvent)
  {
  GrafPtr    oldPort;
  WindowPtr  aWindow = (WindowPtr) anEvent->message;
  TWindow  * wobj;

  GetPort(&oldPort);
  SetPort(aWindow);
  BeginUpdate(aWindow);
  if ((wobj = GetWindowObject(aWindow)) != nil)
    wobj->Draw();
  EndUpdate(aWindow);
  SetPort(oldPort);
  }
```

HandleClose

```
void
HandleClose(WindowPtr aWindow)
  {
  short  windowKind;
  TWindow  *wobj;

  if (aWindow)
    {
    windowKind = ((WindowPeek) aWindow)->windowKind;
    if (windowKind < 0)
      {
      CloseDeskAcc(((WindowPeek)aWindow)->windowKind);
      }
    else if ( ((wobj = GetWindowObject(aWindow)) != nil) &&
          wobj->CanClose() && wobj->Close()
          && wobj->DeleteAfterClose())
      {
      delete wobj;
      }
    }
  }
```

DIALOG BOX



By Scott T Boyd, Editor

Unbiased? Or Propaganda?

I suffer from a fundamental worry that by appearing to be neutral in the OLE/COM vs. OpenDoc/SOM war, you as an editor, are doing the Mac community a great disservice.

Clearly, this is a war to determine whether Apple or Microsoft will control the central standards used to create a document-centric environment on the Mac. After a couple of years of seeing Microsoft upgrade Mac products six months to a year after the corresponding Windows product and reflecting for a moment on where Microsoft's future lies, I find it hard to believe that the **principal** Mac developer magazine wouldn't come out four square behind Apple. Independent of which approach is better (more to follow), Apple needs your help and you should be there for them.

I have read the docs from both Apple and Microsoft (and the SOM manual that is flamed in your "unbiased" comparison piece). SOM is good stuff and it exists right now. Sure, OpenDoc is in Alpha, but if you read the documentation and talk to some of the people in the Alpha program, you'd see that the Apple/IBM/WordPerfect approach is better grounded than the Microsoft approach. Microsoft was first with OLE 1.O and that's part of their problem. To be backward compatible with OLE 1.0 they're getting the same hash that resulted from basing Windows on DOS. But they'll improve...that's for sure.

But I will say that their July diatribes were marketing FUD... a lot of crap written with a lawyer on the team. If I were IBM, I'd be fuming.

In your editorial capacity, I realize that you must allow freedom of expression by your contributing authors, but you also have a responsibility to **not** take a Pontius Pilot attitude at a time when your opinion might make a difference. If you really think that Microsoft has Apple's or the industry's best interests at heart, you don't deserve to be occupying the chair that you currently enjoy. If Microsoft controls the Mac document architecture, then it's all over for the Mac and a poor time for the industry (and incidentally for your magazine...which up to now I have admired enormously and would greatly miss).

The "unbiased" article your magazine printed to compare OLE/COM and OpenDoc/SOM appeared to me to be very little more than a crude rehash of the Microsoft propaganda with a C++ bigot's flame on a somewhat C++ hostile SOM.

I don't have all the information to make a completely informed decision yet based solely on bits and bytes. I can, however, tell you what a strategically bad position Apple will be in if Microsoft has the **power** to dictate that OLE/COM be the linking mechanism of choice on the Mac ... and you know it too!

So vote with your support. The Apple approach has merit and deserves support. Without friends, Microsoft will whomp on Apple and the rest of the industry using their installed Windows base as a club. If you think that Microsoft would not abuse their power if they had no competitor in this area, please think again. Microsoft's record proves otherwise. OLE/COM won't go away. I **need** an alternative.

That's why my company will support OpenDoc/SOM and applications that use this standard.

Think. *– Stephen Johnson, Menlo Park, CA*

We're not done yet with our coverage of OpenDoc and OLE. We will bring you hard information on the SOM issue in particular. As I've been talking with those who have been using it, it sounds like SOM is quite cool, and not the burden that Jeff warned it might be. We've been working on getting some Apple OpenDoc experts writing articles for us, but they're so busy working on OpenDoc itself that it's tough to get them to take on any additional responsibilities. So, if you're reading this, and you've written an OpenDoc part, write us a real-life-story article about it. Write us at editorial@xplain.com – Ed stb

Something has to be done

Apple insisted at WWDC that they would do everything they could to get system 7.5 in the hands of as many users as possible. If the pricing in MacWeek is correct, I believe that they have changed their minds.

MacWeek says that a single user version of 7.5 will cost $135. I realize that the street price will be lower, but it won't be low enough. There are a great many users who are still using system 7.0.1 because the $79 upgrade cost was too high. Given this, how can Apple expect end users to pay $135 for system 7.5.

I think something must be done. I think the price of system 7.5 has got to be lower.

We are working on two products which *require* PowerTalk. We made this decision because Apple took the position that they would price system 7.5 to get it into the hands of as many users as possible. If the user base doesn't upgrade, they won't have PowerTalk. If they don't have PowerTalk, they won't buy our products. It's quite possible that we will have to spend months rewriting our software to work without PowerTalk. This is not a prospect that I find exciting.

Why the change in direction? Why isn't system 7.5 priced *lower* than system 7.1 so that as many users as possible upgrade to the new system?

– Howard Shere, President
Green Dragon Creations, Inc., Water Valley, MS
Howard_Shere@GreenDragon.com

The Editor's Page continued from page 4

Hawking His Wares

Stephen Hawking, renowned physicist, gave a thoughtful keynote, taking us through the thought process of wondering whether there is life elsewhere in the universe. Along the way, he identified computer viruses as a life form (controversial, but not convincing), and used it to get a good laugh. His humor may very well have been the highlight of his talk. At a show like MacWorld, it wasn't surprising to discover that a big reason for his keynote was his new CD-ROM multimedia version of *A Brief History of Time*, his 1988 best-seller.

Back in the Real World

Last week I dropped in on a Software Entrepreneurs Forum in Palo Alto for an OpenDoc vs. OLE debate. Both sides brought their heavy hitters, and they played to a packed audience. Every time I go to one of these debates, both side refine their approaches, using lessons learned in previous debates.

I keep finding myself looking at the issues being discussed. Is SOM the right object model? Do developers really need non-rectangular objects? Will the shipping technology "win" because it's available and the other is not? Is a framework necessary, or programming components at the "bare-metal" API level easy enough?

In this issue, we have a letter where the writer insists that MacTech Magazine rally behind Apple, lending our support because we owe our support to Apple, and so Microsoft doesn't win. A debate along these lines is happening in comp.sys.mac.programmer.

I expect that most of you find at least some of these questions interesting, perhaps even worthy of debate. I have to wonder, though, are these really the issues you care about?

For example, how will you, the developer, decide whether you will support OpenDoc? Will it take convincing you that OpenDoc is the superior technology? Maybe knowing that it's available cross-platform will be your deciding factor? Maybe SOM is what you've been waiting for? It could be that you'll go with it because it's an Apple technology, and you'll follow Apple's lead. Or maybe you'll go with it because it's *not* a Microsoft technology.

And how will you decide whether you'll support OLE? Microsoft Office is wildly successful. Perhaps you want to play into that market. Of course, it's shipping, and maybe you have to get to market now, and you simply can't afford to wait. Maybe OLE is enough for your needs, or you've already got a Windows product, and it only makes sense to use the Microsoft technology that you believe will become dominant.

Here's an issue we haven't heard much about. What's the debugging experience going to be like when you're intermingling components from a number of different vendors? Is one technology more conducive to easy debugging than the other?

These and other issues will play into the decisions that countless developers will be making. The shape that the debate has been taking focuses on these issues.

Yet, there's something we haven't heard enough about, and that's the business model. The debate we've seen so far has centered on interesting issues, but it's time that the debate starts hitting on the issues that make the real difference for developers.

The biggest of these: how are you going to make money? A number of questions come to mind, and none of these has received the kind of treatment necessary for business planning at the small, third-party developer level. Will there be room for more than one kind of each component in the marketplace? How does a little developer get into the channels? Will makers of suites have their collection of components that are good enough, leaving little room for better components too gain entry? Will end-users really shop for individual components? Will Apple and Microsoft do something to help the small developer survive the transition?

Here's another big question. Is it really an either/or choice? It's easy to come away from these debates with the notion that you have to choose up sides. It's also easy to come away thinking that you've got to make up your mind soon.

We're not even sure why Apple keeps attending the debates. Maybe it's just a way to get in front of developer audiences. Maybe it's to give developers a reason to stall and not make a big committment to OLE just yet. Apple has a strong offering. It interoperates with OLE. We see little reason to position it as a competitor.

We're going to take the time to examine both technologies and the business issues surrounding them. We expect the vested interests to take issue with some of what gets printed here. That's why we invite all comers to make their case. And feel free to rebutt. But let's get the debate focused on the needs of the developer. By the way, if there's an issue you'd like to see addressed, let us know, and we'll raise it with the players.

Prograph Conference

Prograph International is hosting the 2nd Annual Prograph Developers' Conference October 14-16 at Apple's R&D campus in Cupertino, CA. The conference will focus on new developments in client/server database tools and cross-platform technologies for PowerMac and Windows. (415) 773-8234 for more info on how you can spend three days soaking up Apple atmosphere while immersing yourself in Prograph.

Food For Thought

Ever wonder how Apple comes up with its licensing strategy and pricing? We had to scratch our heads over this one. To license QuickTime Package 2.0 for Macintosh costs $300/year, yet QuickTime for Windows 1.1 is only $250. What's the message here?

Not Food For Thought

You know that glue we used to bind the CD into the August issue? We visited our printer and asked them what it was called. The technical term? Booger glue.

Tip of The Month continued from page 96

```
    PenPat((Pattern *)"\xF0\xF0\xF0\xF0\xF0\xF0\xF0\xF0");
    for (lp = 0; lp < 99; lp++) {
        MoveTo(0, 0);
        if (lp & 1) {
            start = 0;
            inc = 256;
            stop = 256 * 255;
        } else {
            stop = 0;
            start = 256 * 255;
            inc = -256;
        }
        for (i = start; i != stop; i += inc) {
            if (!optimized) {
                theColor.red   = theColor.green
                               = theColor.blue
                               = i;
                RGBForeColor(&theColor);
                theColor.red   = theColor.green
                               = theColor.blue
                               = 65536 - 256 - i;
                RGBBackColor(&theColor);
            } else {
                cwr.port.rgbFgColor.red
                    = cwr.port.rgbFgColor.green
                    = cwr.port.rgbFgColor.blue
                    = i;
                cwr.port.fgColor =
                    ColorIndex[cwr.port.rgbFgColor.red >> 8];
                cwr.port.rgbBkColor.red
                    = cwr.port.rgbBkColor.green
                    = cwr.port.rgbBkColor.blue
                    = 65536 - 256 - i;
                cwr.port.bkColor =
                    ColorIndex[cwr.port.rgbBkColor.red >> 8];
            }
            Line(100, 0);
            Move(-100, 1);
        }
    }
    stopT = TickCount();
    if (!optimized) {
        *(long *)0x40 = stopT - startT;
    } else {
        *(long *)0x44 = stopT - startT;
    }
}
DebugStr("\p;dm 40");
CloseWindow(wp);
}
```

– Jörg 'jbx' Brown
San Francisco, CA

Tips & Tidbits continued from page 96

they are kept as quasi-standalone applications inside the Translators folder (located in the Symantec C++ folder). This goes for the C, C++, and rez compilers, as well as the .o converter. When the user tells Think Project Manager to compile a file, TPM looks at the file's extension (such as .cp for a C++ file), and launches the appropriate compiler (or translator, as Symantec calls them). This is documented in the Symantec manuals.

What isn't documented, however, is the process by which TPM launches the translators. As it is shipped from the factory, when TPM is launched, it turns around and launches the C++ translator and keeps it in memory. The C translator is left on the disk, and called when necessary. This works great if you do most of your work in C++. But if you're like me, and work mostly in C, this slows down the compilation process because every time a C file is to be compiled, the C translator must launched and loaded into memory, while the C++ translator sits there in memory with nothing to do.

What's a C user to do? Simply modify the C and C++ translators to work the way you want them to. Using ResEdit (or resource editor of your choice), open 'INFO' resource number 0 in the translator named Think C (Symantec graciously includes a ResEdit template for this). Change the setting of Memory Resident Translator? from 0 to 1. Close and save your changes. Voila! The C translator will now be loaded and kept in memory by TPM when it is launched. In one simple step, C compiles are now speeded up. The more files that are being compiled, the greater the speed increase. For instance, a one-file compile is not speeded up much, but if you're working with large projects, the speed increase can be significant.

If you want to free up some memory, you can also modify the C++ translator (named Symantec C++) to not be memory resident.

– Chris Hawk
San Francisco, CA

Dilbert *by Scott Adams*

E-Mail: ScottAdams@aol.com

NEWSBITS

By Scott T Boyd, Editor

New Development Toys

CodeWarrior CW4 has some new features. You can expand and collapse segments in a project window, and add resource files and PowerPlant Constructor documents. CW4 has improved integration with the Metrowerks debugger, and you can debug PowerPC shared libraries, 68k code resources,and 68k threads.

CW4 includes Pascal for PowerPC (1.0dr1) and Universal Pascal interfaces for both 68k and PowerPC. C/C++ compiler goes faster in less memory (e.g. MacApp 3.1 links in half the memory), has inline function support in precompiled headers, and new pragmas.

CW4 has "intrinsic" functions to generate special floating point and low-level synchronization PowerPC assembly instructions PowerPlant now comes as a shared library which you may distribute free of charge. "Attachment" classes change the runtime behavior of objects.

Constructor adds support for tab and radio button groups, internal templates for user-defined data fields, and non-window views and printouts. Profiler is improved, and ZoneRanger is now included. CW4 has 500 additional pages of online documentation. For more info, contact custservice@xplain.com, or call 310/575-4343.

Corruption, Leaks, and More!

The Memory Mine™, a new $99 tool for Macintosh and PowerPC that lets programmers find elusive problems with memory management such as memory leaks and heap corruption much faster than before, is now available. The tool dynamically but non-intrusively monitors use of memory in any open application and shows heap inconsistency and memory leaks as they happen.

In addition to monitoring, The Memory Mine lets you easily stress test memory management in an application, with features to allocate memory in a heap, zap free space, and purge and compact memory. It's a stand-alone tool: no trap patching, and nothing is inserted in code. An application can be monitored as a whole with no need for source code. Or, used with a debugger to set breakpoints in code under development, you can monitor and test sections of code.

It can be used to tune the start-up size of any application to a user's work habits. The Memory Mine runs native on the PowerPC, and needs a Mac with a 68020 or better and System 7. To order The Memory Mine, or for more information, call Adianta Inc. at (408) 354-9569 voice, (408) 354-4292 fax, Applelink ADIANTA, AOL: Adianta.

Low-Hanging Figs

ScriptLink, the product that lets your Newton send AppleScript messages to any scriptable application on an AppleTalk network, is now shipping from Creative Digital Systems. Subscriber price is $425.

Electric PIE Developers 2.45 has about 12 pages of PDA developer-related news, a writeup on the Wireless Datacomm conference that took place in San Jose a few weeks ago and Paul Pott's MacHack '94 epic. All in all, 20 pages of high density information for PDA developers. There is no overlap between the printed and electric versions of the magazine. It's available at ftp://ftp.netcom.com/pub/cds. For more info, call (415) 621-4252 voice, (415) 621-4922 fax, or e-mail cds@netcom.com.

NS BASIC for the Newton

NS BASIC, an implementation of the well-known BASIC programming language, is available for Newton. The product is aimed at business, educational and scientific marketplaces.

You can write programs on the Newton without a host system. The environment is completely interactive. A full complement of functions and data types is provided. Handwritten input, windowing and buttons are supported. Applications can create their own files or access built-in system information, such as Addresses, Notes and Calendar entries.

It includes a 150 page user manual full of examples. Sample code can also be downloaded.

NS BASIC is available through the MacTech Mail Order Store for $99.00. For more information from NS BASIC Corporation, write or call them at 77 Hill Crescent, Toronto, Canada M1M 1J3, (416) 265-5999), Fax: (416) 264-5888, or through the internet at gh@hookup.net.

BasicScript

Summit Software Company announces the availability of BasicScript 2.1 Toolkit for Macintosh and BasicScript 2.1 Toolkit for Power Macintosh, which allow you to add scripting language capabilities, with the same syntax as Microsoft's Visual Basic for Applications (VBA), to Macintosh applications.

The BasicScript Toolkits for Macintosh and Power Macintosh consists of the BasicScript Compiler, Runtime, and Script Editor/Debugger. The BasicScript Compiler generates code that will run without recompilation on all platforms supported by BasicScript, including Windows 3.x, 4.0, and NT, Macintosh (including PowerMacs), MS-DOS, UNIX, NetWare and OS/2. The BasicScript Runtime is a high-performance interpreter that end users can distribute without royalties. The BasicScript Script Editor/Debugger provides the end user with an

integrated development environment for scripts.

Get your free evaluation copies of the BasicScript Toolkits for Macintosh and Power Macintosh. Summit Software licenses BasicScript for Macintosh and Power Macintosh to its customers for royalties and/or license fees. Pricing varies.

For more info, e-mail bfisher@summsoft.com, or mail 2844 Sweet Road, Jamesville, NY 13078. (315) 677-9000 voice, (315) 677-3224 fax.

Address Correction

We incorrectly reported an address in our August issue: Water's Edge Software, PO Box 70022, 2441 Lakeshore Road West, Oakville, Ontario, Canada, L6L 6M9

Updated GCS Compressor

Glen Canyon Software updated GCS Compressor, its compression engine for use with Apple's Installer. New features include a scripting language for automating archive building, and a simple method of splitting large compressed files across multiple disks. The scripting language can specify specific files or files within a folder to add to an archive.

$329 licenses a single product, $599 licenses any number. No annual license or per copy charges. Upgrades are $79.

For more info, contact Glen Canyon at 3921 Shasta View, Eugene OR, 97405. AppleLink: GlenCanyon, Internet: gcs!info@efn.org, or by phone at (800) 477-6947 or (503) 345-6360 voice, or 503-345-6503 fax.

Bad News/Good News

Advanced A. I. Systems announced that it's ceasing development of AAIS Full Control Prolog for the Macintosh. The current version, 3.1.3, will be the last. AAIS Prolog has been available on the Mac since 1986.

AAIS is offering a special last chance to own price of $99 (plus $8 S&H) for both the AAIS Full Control Prolog development system and the AAIS Prolog Program Creator runtime-distribution system (originally listed at $794). There is no royalty charge for distribution of applications created using Prolog and the Program Creator. The product includes approx. 720 pages of documentation, and 15,000+ lines of Prolog code.

For more info, contact AAIS at PO Box 39-0360, Mountain View, CA 94039, or by phone at (415) 948-8658 voice, (415) 948-2486 fax, or by e-mail at AAISProlog@aol.com.

4D on the InfoBahn?

ACI announced a new feature in 4D First – graphical USENET newsgroup interface. 4D First is a $99 database which comes with ten ready-to-use (and ready-to-modify) database templates, and it comes with an interface builder. You can search, sort, export, and print, as well as send and receive e-mail, as well as drag and drop things between folders.

For more info, contact ACI US at 20883 Stevens Creek Blvd., Cupertino, CA 95014. (408) 252-4444 voice, (408) 252-4829 fax. AppleLink: D4444.

Object Pascal for PowerMacs

Language Systems shipped their Object Pascal compiler at Macworld Expo in Boston. LS Object Pascal CD (beta release) *[do we detect a trend? – Ed stb]* contains over 100 MB of developer tools and documentation. It includes Pascal compilers for both 68K and PowerPC-based Macintosh computers, interface files, source level debuggers, and MPW tools for creating native applications.

Included is a custom version of AppMaker, from Bowers Development, which creates a complete Mac user interface in minutes, then generates commented Pascal source files automatically. Language Systems is working with an independent co-op of developers to find a solution for a native version of MacApp 2.0.

LS Object Pascal sells for $399, includes e-mail tech support and free monthly updates, including the final release later this year. Developer support package that includes unlimited telephone support and one year of free updates is available separately for $199. Language Systems stands behind their products with a full money-back guarantee. (800) 252-6479, (703) 478-0181 voice, (703) 689-9593 fax, ALink LANGSYS

Power APL

MicroAPL announced a new implementation of APL for the Power Macintosh which is 100% compatible with previous 68K versions (and highly compatible with Manugistics' APL*Plus III, the leading Windows APL). APL Level II includes a new object-based UI builder facility, and has been reengineered to take advantage of the PowerPC processor.

The new APL adds support for QuickTime, has an improved development environment, and improves APL multitasking. Math runs faster, too. For example, integer arithmetic runs about ten times faster. They got these speed improvements by using PortASM, their assembly-language translation tool, to port the APL.68000 interpreter to the PowerPC architecture.

Version 3 of APL Level II is available for $900, and includes both 68K and PowerPC versions of the interpreter. Upgrades cost $200 for 68K only, or $450 for PowerPC. Level I owners can upgrade for the same price until January. For more info, call (201) 307-9099 in the US, and +44 71 922 8866 in the UK.

Taligent Prepares Enterprise for Launch

Apple, *hp*, and IBM announced roll-out plans for 1995. Apple will ship the application environment (TalAE) on PowerPC and PowerOpen, as well as the People, Places and Things human interface. *hp* will ship TalAE on HP-UX, and IBM will deliver TalAE on OS/2 and AIX. Apple and IBM will work to ensure interoperability with OpenDoc and Taligent technologies. In 1996, when Apple migrates to a microkernel-based OS, Apple will host Taligent Object Services when they ship a microkernel OS in 1996, and will start to build in Taligent technologies.

Get to the core of Macintosh development.

J.P. Morgan, a global financial leader, is writing a whole new chapter in Macintosh development.

Few companies, in fact, can match the sheer range of development efforts or the phenomenal pace at which our base of 2,000+ Macs continues to expand and evolve, including a rapid migration to the Power Macintosh. One result is a highly creative climate where smart, independent thinkers can find genuine challenge and cultivate new proficiencies in areas ranging from object-oriented technology and multimedia to data visualization and globally distributed computing.

For developers of all levels — from associates to team leaders — there are immediate opportunities in numerous business areas. Senior members will share their expertise on multiple projects as developers and mentors. Associates will apply their skills to a broad range of projects that include analysis and design, development, code tuning, regression testing, and configuration management.

All contributors to our cutting-edge efforts must have solid ANSI C or Smalltalk programming skills, knowledge of the Macintosh Toolbox and System 7, and experience in a disciplined software engineering environment. Experience with Unix, ParcPlace VisualWorks, Visix Galaxy, Sybase, C++, real-time market data feeds, and distributed computing is a strong asset, but not required.

To realize your full potential at J.P. Morgan, an equal opportunity employer offering performance-based compensation and benefits including profit sharing, flexible benefits, and tuition reimbursement, please send your resume, with salary history, in confidence, to: **Michael H. Fried, Vice President, Dept. MT, J.P. Morgan, 60 Wall Street, New York, NY 10260-0060. Fax: (212) 235-4888. Internet: sysdev@jpmorgan.com**

We will acknowledge only those candidates who meet the criteria outlined above.

JPMorgan

MacTech Magazine is your recruitment vehicle

When you need to fill important positions at your company, MacTech Magazine is the consistent choice of companies across the country for hiring the best qualified Macintosh programmers and developers. Let MacTech Magazine deliver your recruitment message to an audience of over 27,000 qualified computer professionals.

Call Ruth Subrin at 310/575-4343

Do you have used equipment to sell?

Or do you want to purchase used equipment?

Advertise it in the Classified Section of MacTech Magazine. Call Ruth Subrin at 310/575-4343 to find out how to place your ad.

To receive information on any products advertised in this issue, send your request via Internet: productinfo@xplain.com

If you have a CD-ROM drive, then you've *gotta have* every article published in the first 8 years of MacTech Magazine, including all of 1992. Over 900 articles. All 95 issues. All the source code. THINK Reference 2.0 On Location 2.0 Working applications. Full documentation. Demos for developers. Free shareware code.

Plus, Plus, Plus...

FREE Upgrade to next version of CD-ROM!!*

*Any registered user who purchases *All of MacTech Magazine CD-ROM Volume 1-8* after November 1, 1993 will receive a free upgrade to the next version of the All of MacTech Magazine CD-ROM when available.

MacTech MAGAZINE™ Formerly MacTutor
FOR MACINTOSH PROGRAMMERS & DEVELOPERS

Voice: 310/575-4343 • Fax: 310/575-0925
AppleLink: MT.CUSTSVC • CompuServe: 71333,1063
Internet: custservice@xplain.com
America Online & GEnie: MACTECHMAG

More than 500 megs, selected by Macintosh developers for Macintosh developers, all on the new All of MacTech Magazine CD-ROM, Volume 1–8.

In The MacTech Magazine Folder

- Every article published in the first 8 years
- Complete source code & working examples: Assembly, Pascal, C, FORTRAN, BASIC, Lisp, MacScheme, Forth, Neon, MOPS, Yerk, Prograph, Hypercard, Ada, 4th Dimension, MacApp, THINK Class Library
- Complete text of articles including dialogs, custom windows and menus, sounds, viruses, object oriented programming, animation, simulations, XCMDs and XFCNs, parsing, User Interface Design, List Manager, Class Libraries, debugging, product reviews, inits, control panel devices.
- Special ViewR document reader to view, search and print articles.
- And much more!

THINK™ Reference
FREE!
Symantec's THINK Reference 2.0. Complete on line guide to Inside Macintosh, Vol. I-VI, with cross referenced index, detailed information of each function, procedure and detail needed when programming the Macintosh.
SYMANTEC.™

In The Apple Folder

- "Up Your FCBs"
- BitMapToRegion
- CD-ROM 3.2
- Discipline
- Exec
- MacsBug 6.2.2
- Basic Connectivity Set w/Communications Toolbox
- System 7.0.1 with Tune-up
- Logic Manager 1.0d2
- ResEdit 2.1.1+
- QuickTime 1.5
- And much more!

FREE! ON LOCATION™
A full working, registered version of On Location 2.0, from On Technology. Use it to index all your source code disks. Find references in seconds on multiple disks whether or not the disk is mounted.

Demos relevant to developers

- Demos for all kinds of third party tools and utilities

Public/Shareware Code

- 150+ meg, most are fully documented, including many updates
- Best of BMUG's Programming Tools, System 7 Utilities, etc.
- Hundreds of code ideas, explanations, help files
- Tutorials
- Source code for more than 70 working applications
- And much, much more!

More! More! More!

We could go on listing everything that is included on this new exciting CD-ROM. But you should see it for yourself. Order yours today, for $299 (upgrades for $150) plus shipping & handling. Call, fax, or e-mail for more information.

MACTECH MAGAZINE PRODUCTS & ORDER INFORMATION

E-mail, Fax, write, or call us. You may use your VISA, MasterCard or American Express; or you may send check or money order (in US funds only):
MacTech Magazine, P.O. Box 250055, Los Angeles, CA 90025-9555.
Voice: 310/575-4343 • Fax: 310/575-0925

If you are an e-mail user, you can place orders or contact customer service at:
- **AppleLink:** MT.CUSTSVC
- **CompuServe:** 71333,1063
- **Internet:** custservice@xplain.com
- **America Online:** MT CUSTSVC
- **GEnie:** MACTECHMAG

SUBSCRIPTIONS
US Magazine: $47 for 12 issues
Canada: $59 for 12 issues
International: $97 for 12 issues
Domestic source code disk: $77 for 12 issues
Int'l source code disk: $97 for 12 issues

CD-ROM
All of MacTech Volumes I-VIII CD-ROM: Includes over 900 articles from all 95 issues (1984-1992) of MacTech Magazine (formerly MacTutor). All article text and source code. Articles and code are indexed by On Location. The CD includes Symantec's THINK™ Reference 2.0, On Technology's On Location™ 2.0, working applications with full documentation, product demos for developers and more. See advertisement, this issue: $299. Upgrades $150, e-mail, call or write for info.

BOOKS
The Best of MacTutor, Volume 1: **Sold Out**
The Complete MacTutor, Volume 2: **Sold Out**
The Essential MacTutor, Volume 3: $19.95
The Definitive MacTutor, Volume 4: $24.95
The Best of MacTutor, Volume 5: $34.95
Best of MacTutor Collection, Volumes 3 – 5: $69
Best of MacTutor, Volumes 6, 7, 8 & 9: Not available

DISKS
Source Code Disks: $8 each
Topical Index (1984-1991) on disk: $5

MAGAZINE BACK ISSUES
Volumes 3, 4, 5, 6, 7, 8, 9 and 10: $5 each (subject to availability)

SHIPPING, HANDLING & TAXES
California:
Source disk or single issue: $3
Single book or multiple back issues: $5
Two books: $8 • All other orders: $12

California residents include 8.25% sales tax on all software, disks and books.

Continental US:
Source disk or single issue: $3
Single book or multiple back issues: $7
Two books: $15 • All other orders: $17

Canada, Mexico and Overseas: Please contact us for shipping information.

Allow up to 2 weeks for standard domestic orders, more time for international orders.

PLEASE NOTE
Source code disks and journals from MacTech Magazine are licensed to the purchaser for private use only and are not to be copied for commercial gain. However, the code contained therein may be included, if properly acknowledged, in commercial products at no additional charge. All prices are subject to change without notice.

10% OFF ALL BOOKS! 3RD PARTY PRODUCTS

MACTECH EXCLUSIVES

MacTech Magazine is your exclusive source for these specific products:

NEW! Ad Lib 2.0 The premier MacApp 3.0 compatible ViewEdit replacement. A powerful user-interface editing tool to build views for MacApp 3.0 and 3.1. Ad Lib allows subclassing of all of MacApp's view classes including adorners, behaviors, and drawing environments. String and text style resources are managed automatically. Alternate display methods, such as a view hierarchy window, allow easy examination of complex view structures. Ad Lib includes source code for MacApp extensions that are supported by the editor – buttons can be activated by keystrokes, behaviors can be attached to the application object, and general purpose behaviors can be configured to perform a number of useful functions. Run mode allows the user to try out the views as they will work in an application. Templates can be created to add additional data fields to view classes. Editing palettes provide fast and easy editing of common objects and attributes. Works with ACI's Object Master (version 2.0 and later) to navigate a project's user interface source code. $195

NEW! MAScript 1.2 adds support for AppleScript to your MacApp 3.0.1 and 3.1 based applications. Make your application scriptable and recordable by building on a tried and tested framework for object model support. MAScript dispatches Apple events to the appropriate objects, creates object specifiers, and makes framework objects like windows and documents scriptable and recordable. Sample application shows you how to begin adding support for scripting and recording. MAScript includes complete source code. Install MAScript by modifying one MacApp source file, then adding another to your project. Future versions of MacApp will incorporate MAScript, so MAScript support you add now will work in the future. $199

NEW! Savvy OSA support includes attachability, recordability, scriptability, coercion, in addition to script execution, idling and i/o. Apple event support includes complex object specifiers, synchronous/asynchronous Apple event handling, and Apple event transactions for clients and servers. The Core Suite of Apple event objects is supporting including the application, documents, windows, and files. Documentation includes technology overview, cookbook, and sample code. $250

NEW! More Savvy includes all Savvy features plus Apple event support for all sub-classes of TEventHandler with extensive view support. Apple event support for text includes text attributes and sub-range specification. Recordability supports additional actions, and coercion includes additional types. Additional client and server Apple events. $450

NEW! Super Savvy includes all More Savvy features plus compile, edit, and record scripts using built in script editor. View template editors, like Ad Lib, can attach scripts to view objects and modified scripts are saved with the document. Script action behavior allow quick access for executing and editing scripts attached to views. Text to object specifier coercion plus more. $700

NEW! Savvy QuickTime Requires Savvy, More Savvy, or Super Savvy. Includes QuickTime, Apple event and view template support. Movies come out of the box ready to play, edit, and react to Apple events. They can be included in any view structure, including templates, and are displayed in the scrap view. Movie controls include volume, play rate, looping mode, display style, and other characteristics. $250

NEW! Savvy DataBase Requires Savvy, More Savvy, or Super Savvy. Available Winter 1994. $250

MacTech Magazine is your exclusive source for available back issues of SFA's magazine, source code disks and assorted CD's. Call for more info and pricing.

BOOKS

Defying Gravity: The Making of Newton Doug Menuez and Markos Kounalakis. An in depth, dramatic account of the story of Newton's creation. It is a technological adventure story; a fascinating case study of the process by which an idea is born and then translated into a product on which careers and fortunes can be made or lost. It is a new kind of business book, one that captures through powerful photo-journalism and a fast-paced text, the human drama and risk involved in the

invention of a new technology for a new marketplace. 196 pgs., ~~$29.95~~ **$26.95**

The Elements of E-Mail Style by Brent Heslop and David Angell. Learn the rules of the road in the e-mail age. Concise, easy-to-use format explaining essential e-mail guidelines and rules. It covers style, tone, typography, formatting, politics and etiquette. It also outlines basic rules of composition within the special context of writing e-mail and includes samples and templates for writing specific types of e-mail correspondence. 208 pages. ~~$14.95~~ **$13.45**

NEW! **E-Mail Essentials** by Ed Tittel & Margaret Robbins is a hands-on guide to the basics of e-mail, the ubiquitous networks communication system. The book is suitable for both the casual e-mailer and the networking professional, as it covers everything from the installation of e-mail to the maintenance and management of e-mail hubs and message servers. The books explains the fundamental concepts and technologies of electronic mail, featuring chapters on Lotus applications and CompuServe, as well as information on upgrading, automation, message-based applications, and user training. E-mail is a step-by-step, jargon-free guide that will enable the e-mail user to get the most out of the communication potentials of networking. A Publication of AP Professional September 1994, paperback, 250 pp. ~~$24.95~~ **$22.45**

NEW! **Graphics Gems IV** edited by Paul Heckbert Volume IV is the newest collection of carefully crafted, innovative gems. All of the gems are immediately accessible and useful in formulating clean, fast, and elegant programs. The C programming language is used for most of the program listings, although several of the gems have C++ implementations. An IBM or Macintosh disk containing all of the code from all four volumes is included. April 1994, Hardcover, 512 pp A Publication of AP Professional Includes one 3.5" high-density disk. ~~$49.95~~ **$44.95**

How To Write Macintosh Software by Scott Knaster is a great source for understanding Macintosh programming techniques. Drawing from his years of experience working with programmers, Scott explains the mysteries and myths of Macintosh programming with wit and humor. The third edition, fully revised and updated, covers System 7 and 32-bit developments, and explores such topics as how and where things are stored in memory; what things in memory can be moved around and when they may be moved; how to debug your applications with MacsBug; how to examine your program's code to learn precisely what's going on when it runs. 448 pgs., ~~$28.95~~ **$26.05**

The Instant Internet Guide by Brent Heslop and David Angell. An Internet jump-start – how to access, use and navigate global networks. The Instant Internet Guide equips readers with the tools needed to travel the electronic world. The book highlights the most important sources of Internet news and information and explains how to access information on remote systems. It outlines how to use essential Internet utilities and programs and includes a primer on UNIX for the Internet. 224 pages ~~$14.95~~ **$13.45**

Learn C on the Macintosh by Dave Mark. This self-teaching book/disk package gives you everything you need to begin programming on the Macintosh. Learn to write, edit, compile, and run your first C programs through a series of over 25 projects that build on one another. The book comes with THIN C – a customized version of Symantec's THINK C, the leading programming environment for Macintosh. 464 pages, Book/disk: ~~$34.95~~ **$31.45**

Learn C++ on the Macintosh by Dave Mark. After a brief refresher course in C, Learn C++ introduces the basic syntax of C++ and object programming. Then you'll learn how to write, edit, and compile your first C++ programs through a series of programming projects that build on one another as new concepts are introduced. Key C++ concepts such as derived classes, operator overloading, and iostream functions are all covered in Dave's easy-to-follow approach. Includes a special version of Symantec C++ for Macintosh. Book/disk package with 3.5" 800K Macintosh disk. 400 pages, ~~$36.95~~ **$33.26**

Macintosh C Programming Primer Volume I, Second Edition, Inside the Toolbox Using THINK C by Dave Mark and Cartwright Reed. This new edition of this Macintosh programming bestseller is updated to include recent changes in Macintosh technology, including System 7, new versions of THINK C and ResEdit, and new Macintosh machines. Readers will learn how to use the resources, Macintosh Toolbox and interface to create stand-alone applications. 672 pages, ~~$26.95~~ **$24.25**

Macintosh C Programming Primer Volume II, Mastering the Toolbox Using THINK C by Dave Mark. Volume II picks up where Volume I leaves off, covering more advanced topics such as: Color QuickDraw, THINK Class Library, TextEdit, and the Memory Manager: 528 pgs. ~~$26.95~~ **$24.25**

Macintosh Pascal Programming Primer Volume I, Inside the Toolbox Using THINK Pascal by Dave Mark and Cartwright Reed. This tutorial shows programmers new to the Macintosh how to use the Toolbox, resources, and the Macintosh interface to create stand-alone applications with Symantec's THINK Pascal. 544 pages ~~$26.95~~ **$24.25**

Macintosh Programming Techniques by Dan Sydow (Series Editor: Tony Meadow). This tutorial and handbook provides a thorough foundation in the special techniques of Macintosh program-ming for experienced Macintosh programmers as well as those making the transition from DOS, Windows, VAX or UNIX. Emphasizes programming techniques over syntax for better code, regardless of language. Guides the reader through Macintosh memory management, QuickDraw, events and more, using sample program in C++. Disk includes an interactive tutorial, plus reusable C++ code. ~~$34.95~~ **$31.95**

NEW! **Multimedia Authoring Building and Developing Documents** by Scott Fisher addresses the concerns that face anyone trying to create multimedia documents. It offers specific advice on when to use different kinds of information architecture, discusses the human-factors concepts that determine how readers use and retain information, and them applies these findings to multimedia documents, covering the high-level issues concerning planners and authors of multimedia documents as well as those involved in evaluating or purchasing multimedia platforms. A Publication of AP Professional February 1994, Paperback, 320 pp.Includes one 3.5" high-density disk. ~~$34.95~~ **$31.45**

NEW! **Programming for the Newton Software Development with NewtonScript** by Julie McKeehan and Neil Rhodes. Foreword by Walter R. Smith. Programming for the Newton: Software Development with NewtonScript is an indispensable tool for Newton programmers. Readers will learn how to develop software for the Newton on the Macintosh from people that developed the course on programming the Newton for Apple Computer. The enclosed 3.5" disk contains a sample Newton application from the books, as well as demonstration version of Newton Toolkit (NTK), Apple Computers complete development environment for the Newtons. A Publication of AP Professional May 1994, Paperback, 393 pp. ~~$29.95~~ **$26.95**

Programming in Symantec C++ for the Macintosh by Judy May and John Whittle. This book will introduce you to object-oriented programming, the C++ language, and of course Symantec C++ for the Macintosh. You don't have to be a programmer, or even know anything about programming to benefit from this book. Programming in Symantec C++ for the Macintosh covers everything from the basics to advanced features of Symantec C++. If you are a Think C or Zortech C++ programmer who wants to learn more about object-oriented programming or what's different about Symantec C++, there are whole chapters specifically for you. Includes helpful examples of C++ code that illustrate object-oriented programs. ~~$29.95~~ **$26.95**

Programming for System 7 by Gary Little and Tim Swihart, is a hands-on guide to creating applications for System 7. It describes the new features and functions of the operating system in detail. Topics covered include file operations, cooperative multitasking, Balloon Help, Apple events, and the File Manager. Numerous working C code examples show programmers how to take advantage of each of these features and use them in developing their applications. 384 pages ~~$26.95~~ **$24.25**

ResEdit™ Complete, Second Edition by Peter Alley and Carolyn Strange. With ResEdit, Macintosh programmers can customize every aspect of their interface form creating screen backgrounds and icons to customizing menus and dialog boxes. 608 pages. Book/disk package. ~~$34.95~~ **$31.45**

Sad Macs, Bombs, Disasters and What to Do About Them by Ted Landau comes to the rescue with your Macintosh problems. From fractious fonts to the ominous Sad Macintosh icon, this emergency handbook covers the whole range of Macintosh problems: symptoms, causes, and what you can do to solve them. 640 Pages ~~$24.95~~ **$22.45**

Software By Design: Creating User Friendly Software by Penny Bauersfeld (Series Editor: Tony Meadow). This excellent reference provides readers with a thorough how-to for designing software that is easy to learn, comfortable to operate and that inspires user confidence. Written from the perspective of Macintosh, but compatible with all platforms. Stresses user input from initial design, through prototyping, testing and revision. Provides tools for analyzing user needs and test responses. Includes exercises for sharpening user-oriented design skills. ~~$29.95~~ **$26.95**

Writing Localizable Software for the Macintosh by Daniel R. Carter. 469 pages. ~~$26.95~~ **$24.25**

THE APPLE LIBRARY

HyperCard Stack Design Guidelines by Apple Computer, Inc. is an essential book for everyone who creates Apple HyperCard stacks, from beginners to commercial developers. It covers the basic principles of design that, when incorporated, make HyperCard stacks effective and usable. Topics include guidelines,

Want more product info? Call us at 310/575-4343.

navigation, graphic design and screen illustration, text in stacks, music and sound, a sample stack development scenario, collaborative development, and the Stack Design Checklist. 240 pages, ~~$21.95~~ **$19.95**

Inside AppleTalk by Gursharan S. Sidhu, Richard F. Andrews and Alan B. Oppenheimer. Apple Computer, Inc. 650 pages, ~~$34.95~~ **$31.45**

NEW! Inside Macintosh: AOCE Application Interfaces by Apple Computer, Inc. shows how your application can take advantage of the system software features provided by PowerTalk system software and the PowerShare collaboration servers. Nearly every Macintosh application program can benefit from the addition of some of these features. This book shows how you can add electronic mail capabilities to your application, write a messaging application or agent, store information in and retrieve information from PowerShare and other AOCE catalogs, add catalog-browsing and find-in-catalog capabilities to your application, write templates that extend the Finder's ability to display information in PowerShare and other AOCE catalogs, add digital signatures to files or to any portion of a document, and establish an authenticated messaging connection. ~~$40.45~~ **$36.40**

NEW! Inside Macintosh: AOCE Service Access Modules by Apple Computer, Inc. describes how to write a software module that gives users and PowerTalk-enabled applications access to a new or existing mail and messaging service or catalog service. This book shows how to write a catalog service access module (CSAM), a messaging service access module (MSAM), and AOCE templates that allow a user to set up a CSAM or MSAM and add addresses to mail and messages. ~~$26.95~~ **$24.25**

NEW! Inside Macintosh: Devices by Apple Computer, Inc. describes how to write software that interacts with built-in and peripheral hardware devices. With this book,. you'll learn how to write and install your own device drivers, desk accessories, and Chooser extensions; communicate with device drivers using the Device Manager; access expansion cards using the Slot Manager; control SCSI devices using SCSI Manager 4.3 or the original SCSI Manager; communicate directly with Apple Desktop Bus devices; interact with the Power Manager in battery-powered Macintosh computers; and communicate with serial devices using the Serial Driver. ~~$29.95~~ **$26.95**

Inside Macintosh: Files by Apple Computer, Inc. describes the parts of the operating system that allow you to manage files. It shows how your application can handle the commands typically found in a File menu. It also provides a reference to the File and Alias Managers, the Disk Initialization and Standard File Packages. 510 pgs, ~~$29.95~~ **$26.95**

Inside Macintosh: Interapplication Communication by Apple Computer, Inc. shows how applications can work together. How your application can share data, request information or services, allow the user to automate tasks, communicate with remote databases. ~~$34.95~~ **$31.45**

Inside Macintosh: Imaging by Apple Computer, Inc. covers QuickDraw and Color QuickDraw. The book includes general discussions of drawing and working with color. It describes the structures that hold images and image information, and the routines that manipulate them. It also covers the Palette, Color, and Printing Managers, and the Color Picker, Color Matching, and Picture Utilities. ~~$26.95~~ **$24.25**

Inside Macintosh: Macintosh Toolbox Essentials by Apple Computer, Inc. covers the heart of the Macintosh. The toolbox enables programmers to create applications consistent with the Macintosh "look and feel". This book describes Toolbox routines and shows how to implement essential user interface elements, such as menus, windows, scroll bars, icons and dialog boxes. 880 pages ~~$34.95~~ **$31.45**

Inside Macintosh: More Macintosh Toolbox by Apple Computer, Inc. covers other Macintosh features such as how to support copy and paste, provide Balloon Help, play and record sound and create control panels are covered in this volume. The managers discussed include Help, List, Resource, Scrap and Sound. ~~$34.65~~ **$31.45**

Inside Macintosh: Memory by Apple Computer, Inc. describes the parts of the Macintosh operating system that allow you to manage memory. It provides detailed strategies for allocating and releasing memory, avoiding low-memory situations, reference to the Memory Manager, the Virtual Memory Manager, and memory-related utilities. 296 pages, ~~$24.95~~ **$22.45**

Inside Macintosh: Networking by Apple Computer, Inc. describes how to write software that uses AppleTalk networking protocols. It describes the components and organization of AppleTalk and how to select an AppleTalk protocol. It provides the complete application interfaces to all AppleTalk protocols, including ATP (AppleTalk Transaction Protocol), DDP (Datagram Delivery Protocol), and ADSP (AppleTalk Data Stream Protocol), among others. ~~$29.95~~ **$26.95**

Inside Macintosh: Operating System Utilities by Apple Computer, Inc. describes parts of the Macintosh Operating System that allow you to manage various low-level aspects of the operating system. Everyone who programs the Macintosh should read this book! It will show you in detail how to get information about the operating system, manage operating system queues, handle dates and times, control the settings of the parameter RAM, manipulate the trap dispatch table, and receive and respond to low-level system errors. ~~$26.95~~ **$23.45**

Inside Macintosh: Overview by Apple Computer, Inc. is the first book that people who are unfamiliar with Macintosh programming should read. It gives an overview of Macintosh programming fundamentals and a road map to the New Inside Macintosh library. Inside Macintosh: Overview also covers various programming tools and languages, compatibility guidelines and an overview of considerations for worldwide development. 176 pages, ~~$22.95~~ **$20.65**

Inside Macintosh: PowerPC Numerics by Apple Computer, Inc. describes the floating-point numerics environment provided with the first release of PowerPC processor-based Macintosh computers. The numerics environment conforms to the IEEE standard 754 for binary floating-point arithmetic. This book provides a description of that standard and shows how RISC Numerics compiles with it. This book also shows programmers how to create floating-point values and how to perform operations on floating-point values in high-level languages such as C and in PowerPC assembly language. ~~$28.95~~ **$26.00**

Inside Macintosh: PowerPC System Software by Apple Computer, Inc. describes the new process execution environment and system software

Want more product info? E-mail us at productinfo@xplain.com

MAIL ORDER STORE

services provided with the first version of the system software for Macintosh on PowerPC computers. It contains information you need to know to write applications and other software that can run on the PowerPC. PowerPC System Software shows in detail how to make your software compatible with the new run-time environment provided on PowerPC-based Macintosh computers. It also provides a complete technical reference for the Mixed Mode Manager, the Code Fragment Manager, and the Exception Manager. ~~$24.95~~ **$22.45**

Inside Macintosh: Processes by Apple Computer, Inc. describes the parts of the Macintosh operating system that allow you to control the execution of processes and interrupt tasks. It shows in detail how you can use the Process Manager to get information about processes loaded in memory. It is also a reference for the Vertical Retrace, Time, Notification, Deferred Task, and Shutdown Managers. 208 pages, ~~$22.95~~ **$20.65**

Inside Macintosh: QuickTime by Apple Computer, Inc. is for anyone who wants to create applications that use QuickTime, the system software that allows the integration of video, animation, and sounds into applications. This book describes all of the QuickTime Toolbox utilities. In addition, it provides the information you need to compress and decompress images and image sequences. ~~$29.95~~ **$26.95**

Inside Macintosh: QuickTime Components by Apple Computer, Inc.covers how to use and develop QuickTime components such as image compressors, movie controllers, sequence grabbers, and video digitizers. ~~$34.95~~ **$31.45**

Inside Macintosh: Sound by Apple Computer, Inc. describes the parts of the Macintosh system software that allow you to manage sounds. It contains information that you need to know to write applications and other software that can record and play back sounds, compress and expand audio data, convert text to speech, and perform other similar operations. ~~$26.95~~ **$24.25**

Inside Macintosh: Text by Apple Computer, Inc. describes how to perform text handling, from simple character display to multi-language processing. The Font, Script, Text Services, and Dictionary Managers are all covered, in addition to QuickDraw Text, TextEdit, and International and Keyboard Resources. ~~$39.95~~ **$35.95**

Inside Macintosh: QuickDraw™ GX Library by Apple Computer, Inc. is the powerful new graphics architecture for the Macintosh. Far more than just a revision of QuickDraw, QuickDraw GX is a unified approach to graphics and typography that gives programmers unprecedented flexibility and power in drawing and printing all kinds of shapes, images, and text. This long-awaited extension to Macintosh system software is documented in a library of books that are themselves an extension to the new Inside Macintosh series. The QuickDraw GX Library is clear, concise, and organized by topic. The books contain detailed explanations and abundant programming examples. With extensive cross-references, illustrations, and C-language sample code, the QuickDraw GX Library gives programmers fast and complete reference information for creating powerful graphics and publishing applications with sophisticated printing capabilities. The first two volumes in the QuickDraw GX Library are:

Inside Macintosh: QuickDraw GX Objects by Apple Computer, Inc. introduces QuickDraw GX and its object structure, and shows programmers how to manipulate

objects in all types of programs. ~~$26.95~~ **$24.25**

Inside Macintosh: QuickDraw GX Graphics by Apple Computer, Inc. shows readers how to create and manipulate the fundamental geometric shapes of QuickDraw GX to generate a vast range of graphic entities. It also demonstrates how to work with bitmaps and pictures, and specialized QuickDraw GX graphic shapes. ~~$26.95~~ **$24.25**

LANGUAGES

New Pricing! **CodeWarrior™ CD** by Metrowerks comes in two versions – Bronze and Gold. These CDs contain the CodeWarrior development environment including C++, C and Pascal compilers; high-speed linkers; native-mode interactive debuggers; and a powerful new application framework called PowerPlant for rapid Macintosh development in C++. Bronze generates 680x0 code. Gold generates both 680x0 and PowerPC code. All versions are a 3 CD subscription over a 1-year period. Bronze: $99, Gold: $399. **Bronze comes with a 6-month MacTech subscription. Gold comes with a 1-year subscription. Both at no additional charge!**

NEW! **Geekware** by Metrowerks is here! In high school, they called you a computer geek. Now, they work at burger joints and wear polyester uniforms. And you don't. Wear it to your favorite burger joint. $24.95

FORTRAN by Language Systems is a full-featured ANSI standard FORTRAN 77 compiler that runs in the Macintosh Programmers Workshop (MPW). All major VAX extensions are supported as well as all major features of Cray and Data General FORTRAN. FORTRAN creates System 7 savvy applications quickly and easily. Compiler options specify code generation and optimization for all Macintoshes, including special optimizations for 68040 machines. Error messages are written in plain English and are automatically linked to the source file. The runtime user interface of compiled FORTRAN programs is fully customizable by programmers with any level of Macintosh experience. $595. w/o MPW: $495. Corporate 5 pack $1575

FORTRAN 77 SDK for Power Macintosh by Absoft includes a globally optimizing native compiler and linker, native Fx™ multi-language debugger, and Apple's MPW development environment. The compiler is a full ANSI/ISO FORTRAN 77 implementation and includes all MIL-STD 1753 extensions, Cray/Sun-style POINTER, and several Fortran 90 enhancements. MRWE, Absoft's application framework libraries, is included as is the MIG graphics library for quick creation of plots and graphs. The native Macintosh PPC toolbox is fully supported. Absoft's Fx debugger can debug intermixed FORTRAN 77,C, C++, PPC assembler. The compiler, linker, and debugger all run as native PPC tools and produce native Macintosh PPC executables. $699

MacFortran® II V3.3 is a VAX/VMS compatible, full ANSI/ISO FORTRAN 77 compiler including all MIL-STD 1753 extensions. Acknowledged to be the fastest FORTRAN available for Macintosh, MacFortran II is bundled with the latest version of Macintosh Programmer's Workshop (MPW), and includes SourceBug (Apple's source level symbolic debugger) and SoftwareFPU (a math co-processor emulator). Also included is Absoft's Macintosh Runtime Window Environment (MRWE) application framework (with fully documented source code as examples) and MIG graphics library. MacFortran II v3.3 features improved 68040CPU support and is fully compatible with Power Macintosh under emulation. Documentation includes special sections devoted to use of MacFortran II with the MPW editor and linker, implementation of System 7 features, and porting code to the Macintosh from from various mainframes and Unix workstation platforms. $595

REM GOSUB LET *NS* BASIC

NEW! **BASIC for the Newton** is BASIC for the Newton! From NS BASIC Corporation, it is a fully interactive implementation of the BASIC programming language. It runs entirely on the Newton - no host is required. It includes a full set of functions and data types, hand-written input, windows, buttons and extensions to take advantage of the Newton environment. Applications can create files or access the built-in soups. Applications can also access the serial port for input and output. Work directly on the Newton, or through a connected Mac/PC and keyboard. NS BASIC includes a 150 page pocket sized manual. $99

SmalltalkAgents™, a superset of the Smalltalk language, is fully integrated with Macintosh, incorporating design features specifically for the RISC and Macintosh System 7 architecture. SmalltalkAgents is a true object oriented workbench that includes an incremental and extensible compiler, an array of design and cross reference tools, pre-emptive interrupt driven threads and events, an extensive class library including classes for general programming, classes for the Macintosh user interface and classes for the Macintosh operating system. Integration of components in enterprise systems is simplified with the network, telecommunication, and inter-application communication libraries. The SmalltalkAgents' extensive class library and add-on components make it especially well suited as a development workbench for custom applications in business, education, science, engineering, and academic research. $695

SYMANTEC.™

Symantec C++ for Macintosh is an object oriented development environment designed for professional Macintosh programmers. Symantec C++ features powerful object-oriented development tools within a completely integrated environment. The C++ compiler, incremental linker, THINK Class Library, integrated browser, and automatic project management give Symantec C++ fast turnaround times. This product supports multiple editors and translators, so you can use your favorite tools and resource editors as well as scripts you've written within the environment. And with ToolServer, you'll be able to customize menus and attach scripts based on Apple events, AppleScript, and MPW Tools. The built-in SourceServer provides a source code control system, allowing teams of programmers to solve tough problems faster. With SourceServer, you'll always know you're working on the latest version. And you'll have old versions at your fingertips when code "breaks" and you need to look back at modifications. Product Contents: Three high density disks, an 832-page user manual, a 568-page THINK Class Library and a 100-page C++ Compiler Guide. $369

THINK C by Symantec Corporation. THINK C is easy to use and highly visual, making it the No. 1 selling Macintosh programming environment. Enhancements make this product faster and more versatile than ever, improving your productivity with more powerful project management, a full set of tools, and script support for major script-based languages. With the THINK environment, you spend less time on routine programming tasks due to an extremely fast compiler and incremental linker. In addition, the automatic project manager saves you time by tracking changes to your files and recompiling only those that have changes. All the tools you need – a multi-window editor, compiler, linker, debugger, browser, and resource editor – are completely integrated for speed and ease of use. One of the most valuable of these tools is the THINK Class Library, a set of program building blocks that gives you a head start in writing object-oriented applications. And with the new open architecture, you can use your favorite tools, resource editors, and scripts within the environment. THINK C is the logical next step for programmers who have worked in HyperCard or other script-based development environments. The environment supports AppleScript, Apple events, and Frontier, so you can link and automate complex, multi-project operations. Product Contents: Four Macintosh disks, an 832-page user manual, and a 568-page THINK Class Library Guide. $219

THINK Pascal v. 4.0 by Symantec Corporation. Professionals and students will welcome this version of THINK Pascal. It is fully integrated for rapid turnaround time and lets you take advantage of System 7 capabilities. Features include support for large projects, enhanced THINK Class Library, System 7 compatibility, superior code generation, and smart linking. Product Contents: Four Macintosh disks, a 562-page user manual, and a 498-page object-oriented programming manual. $169

UTILITIES

BBEdit 3.0 from Bare Bones Software is now Accelerated for Power Macintosh. This powerful, intuitive text editor offers integrated support for THINK C 7.0, THINK Reference 2.0 and MPW ToolServer. BBEdit's many features include: Integrated PopUpFuncs™ technology for speedy navigation of source code files, unique "Find Differences" command (BBEdit can find differences between projects and folders as well as files), support for Macintosh Drag and Drop for editing and other common tasks, PowerTalk support for reading, sending and composition of PowerTalk mail, scripting via any OSA compatible scripting language including AppleScript and Frontier 3.0, and fast search and replace with optional "grep" matching and multi-file searching.

Want more product info? Call us at 310/575-4343.

BBEdit's robust feature set and proven performance and reliability make it the editor of choice for professionals and hobbyists alike. $99

C Programmer's Toolbox/MPW Rev. 3.0 by MMCAD. The C Programmer's Toolbox provides a wealth of programming and documentation support tools for developers who are creating new code, porting existing code, or trying to improve and expand existing code. The tools include: CDecl composes and translates C/C++ declaration statements to/from English; CFlow™ determines program function hierarchy, runtime library contents, function/file interdependencies and graphs all or part of a program's functional structure; CHilite™ highlights and prints C/C++ files; CLint™ semantically checks multiple C source files, identifying potential programming bugs; CPrint™ reformats, beautifies and documents C/C++ source files; and more... Works with MPW C/C++, THINK C, requires Apple's MPW. $295

CLImate by Orchard Software is a command line interface that lets you communicate with your Macintosh using English commands to create, delete, rename, and move files and folders. It can start applications, format disks, restart your computer, etc. CLImate supplements the Finder. It includes a BASIC interpreter that lets you script your Macintosh without AppleScript. The interpreter includes advanced programming constructs: repeat loops, if/then/else conditionals, subroutine calls, etc... CLImate implements wildcard characters, enabling you to work on groups of files. Use CLImate instead of MPW to manage your projects. CLImate is an application occupying 70K disk space. It comes bundled with sample programs and full documentation. $59.95

CMaster 2.0 by Jersey Scientific installs into THINK C 5 / 6 / 7 and Symantec C++ for Macintosh, and enhances the editor. Use its function popup to select a function and CMaster takes you right to it. Other features include multiple clipboards and markers, a Function Prototyper, and a GoBack Menu which can take you back to previous editing contexts. Almost all features bindable to the keyboard, along over a hundred keyboard-only features like "Add New Automatic Variable." Glossaries, AppleScript and ToolServer support, Macros, and External Tools you create too! $129.95

Cron Manager by Orchard Software implements the UNIX Cron facility. It can open any Macintosh file on a given date and time. By creating an alias, renaming it to the date and time to open, and moving it into the special Cron Events Folder, Cron Manager will open it. Cron Manager is a control panel that creates the special Cron Events Folder inside your System Folder. It is completely transparent to the user. It works like the Startup Items folder, only smarter. It works with any Macintosh file: if you can double-click to start it, Cron Manager can open it. $26.95. Cron Manager bundled with CLImate, $59.95

Dialog Maker by Electric Software Corporation. Migrating from C to C++? Dialog Maker can ease your transition. Dialog Maker is an object-oriented programming library for MPW C++ and Symantec C++ (MPW and Symantec Development Environment versions) which contains a complete set of routines that create a high level interface to dialogs. Dialog Maker provides a small number of simple, yet powerful routines to access and manipulate dialogs. Resources are used to control the most common dialog behavior allowing you to develop your application lightning fast. Minimum requirements System 7.0, MPW 3.2, MPW C++ 3.2, or Symantec C++ 6.0. $149

InstallerPack™ by StepUp Software is a package of several Installer "atoms" that let developers incorporate graphics, sounds, file compression and custom folder icons into installation scripts. Compression formats supported are Compact Pro & Diamond. Each atom also available separately: $219

NEW! Last Resort Programmer's Edition records every keystroke, command key and mouse event (in local coordinates) to a file on your hard disk. This is especially useful for program testing & debugging, and for technical support and help desks. If something goes wrong (because of a power failure, system crash, forgetting to save or deleting lines) and you lose a word, phrase, or document you can look in the Last Resort keystroke file and recover what you typed. Last Resort is also useful for technical support personnel, when they have to ask "What was the last thing you did before...?" $74.95

NEW! LS Object Pascal CD includes the world's first Object Pascal compiler for Power Macintosh. 100% compatible with Apple's MPW Pascal, LS Object Pascal combines the best of Apple's native development tools with innovative new technology developed at Language Systems. Compiler options specify 68K or native PowerPC code generation. Included on the CD are: LS Object Pascal compiler, Universal Pascal Toolbox interfaces, fully loaded MPW 3.3.1, 68K and PowerPC source debuggers, PowerPC assembler, online documentation, Macintosh Tech Notes, and a special version of AppMaker by Bowers Development that generates native Pascal source code. The beta release includes upgrades to v1.0 when it becomes available. $399

NEW! Spellswell 7 1.0.4 is an award-winning, comprehensive, practical spelling checker that works in batch mode or within applications that incorporate the Apple Events Word Services protocol (e.g., Eudora, WordPerfect, Communicate!, and Fair Witness). Spellswell 7 checks for spelling errors as well as common typos like capitalization errors, spaces before punctuation, double double word errors, abbreviation errors, mixed case errors, extra spaces between words, a/an before vowel/consonant, etc... MacTech orders include developer kit with Writeswell Jr., a sample Apple Events Word Services word-processor and its source code. $74.95

MacAnalyst by Excel Software supports software engineering methods including structured analysis, data modeling, screen prototyping, object-oriented analysis, and data dictionary. This language independent tool is used by system analysts and software designers. Demo $79, Product $995

MacAnalyst/Expert by Excel Software supports software engineering methods with the capabilities of MacAnalyst plus state transition diagrams, state transition tables, decision tables and process activation tables. An integrated requirement database provides traceability from requirement statements to analysis or design diagrams, code or test procedures. This tool is well suited to the analysis and design of real-time or requirements driven projects. Demo $79, Product $1595

MacDesigner by Excel Software supports software engineering methods including structured design, object-oriented design, data dictionary and code browsing. This tool is well suited to detailed design or maintenance of software development projects. Demo $79, Product $995

MacDesigner/Expert by Excel Software supports software engineering methods with the capabilities of MacDesigner plus multi-task design. An integrated requirement database provides traceability from requirement statements to design diagrams, code or test procedures. This tool is well suited to design or maintenance of real-time, multi-tasking software projects. Demo $79, Product $1595

MacA&D by Excel Software combines the capabilities of MacAnalyst/Expert and MacDesigner/Expert into a single application. It supports structured analysis and design, object-oriented analysis and design, real-time extensions, task design, data modeling, screen prototyping, code editing and browsing, reengineering, requirement traceability, and a global data dictionary. Demo $149, Product $2995

MacWireFrame by Amplified Intelligence. Create your own virtual reality application with MacWireFrame, a virtual reality application frame work. Includes a complete library of object oriented graphics routines, its own easy to understand application frame work (similar to MacApp or TCL but a lot easier to understand), plus an example application program that lets you start solid modeling right away. Comes complete with fully documented source code. All new purchases will be guaranteed a $49.99 upgrade to the soon to be released, scriptable, MacWireFrame 5.0. Due to the overwhelming response the special price offer has been extended for a little while longer. **Special Offer:** ~~$299.00~~ **$75!!!!**

Marksman by IT Makers graphically creates program's user interface. $125

McCLint™ Rev. 2.2 by MMCAD. McCLint locates questionable C programming constructs, saving you hours by identifying programming mistakes and latent programming bugs. Some of the checks include variable type usage, conditional and assignment statement usage, arithmetic operations in conditional expressions, misplaced semicolons, pointer type coercion, function argument passing (with and without function prototypes), local and global variable initialization and usage, and existence/shape of return statements. McCLint includes a THINK C like, multiple window editor and source code highlighting system in a fully integrated environment. One or more files can be analyzed in an interactive or batch fashion. Works with THINK C (including OOPS), MPW C,... $149.95

McCPrint™ Rev 2.2 by MMCAD. McCPrint reformats and beautifies C and C++ source code in a user specified manner. You can transform code to and from your programming style, making source code easier to read and work with. In addition to code formatting, documentation support aids include source code pagination, line number inclusion and control flow graphing. McCPrint includes a multiple window editor and source code highlighting system in a fully integrated environment. Works with THINK C, MPW C/C++, supports System 7 and 32 bit addressing for use on any system including the Quadras. $99.95

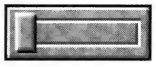
NEW FOR POWERPC AND MACINTOSH

The Memory Mine™ by Adianta is a stand alone debugging tool for Macintosh and native PowerPC. Programmers can monitor heaps, identify problems such as memory leaks, and stress test applications. Active status of

Want more product info? E-mail us at productinfo@xplain.com

memory in a heap is sampled on the fly: allocation in non-relocatable (Ptr), relocatable (Handle) and free space is shown, as are heap corruption, fragmentation, and more... Allocate, Purge, Compact, and Zap memory let users stress test all or part of a program. Source code is not needed to view heaps. It works on Macintoshes with 68020 or later and System 7.0 or later. $99

p1 Modula-2 V5.1 is a full implementation of the ISO Standard for Modula-2 which includes exception handling, termination, complex numbers, value constructors, a standard library and more. In addition it supports objects and MacApp, foreign language calls, all current MPW interfaces, optimized 680x0 instructions, three floating point types with four modes of operation, etc. A symbolic window debugger, several utilities and a set of examples (including MacApp tutorial) are included. p1 Modula-2 requires MPW. It is targeted for professional development and prompt technical support by e-mail or FAX is granted. $395, corporate 5 pack $1175



PictureCDEF by Paradigm Software is a professional-level CDEF for creating custom "puffy" graphical buttons (8-64 pixels in size). PictureCDEF is multi-monitor and bit-depth sensitive. The button graphic (created with ResEdit) can be changed at runtime and even animated with a call-back routine. Create distinct buttons in six variations: PushButton, FlexiButton, ToggleButton, ChkButton, PushPictButton and TogglePictButton. Position the optional button title at left, bottom or right, or follow the system text direction for international support. 25 button frames, manual and sample code included. MacApp 3.0 support. Demo available on request. Full source code: $95. Object code only: $45.

Qd3d/3dPane/SmartPane source code bundle by Vivistar Consulting. **Qd3d 2.0:** Full featured 3d graphics. Points; lines; polygons; polyhedra; Gouraud shading; z-buffering; culling; depth cueing; parallel, perspective, and stereoscopic projections; performance enhancing "OnlyQD" and "Wireframe" modes; full clipping; pipeline access; animation and model interaction support; and a "triad mouse" to map 2d mouse movement to 3d. **3dPane 2.0:** Integrates Qd3d with the TCL and provides a view orientation controller. **SmartPane:** Offscreen image buffering, flicker free animation, and QuickTime movie recording. For use with Qd3d/3dPane or in 2d settings. All work with C++ compilers or ThinkC 6. $192

NEW! **QC™** by Onyx Technology, is a system extension that stress tests code during runtime for common and not-so-common errors. Tests include heap checks, purges, scrambles, handle/pointer validation, dispose/release checks, write to zero, de-reference zero as well as other tests like free memory invalidation and block bounds checking. QC is extremely user friendly for the non-technical tester yet offers an API for programmers who want precise control over testing. $99

QUED/M 2.6 by Nisus is the text editor that has become the industry standard for speed and efficiency. This 32-bit clean version fully supports the MPW ToolServer through Apple Events, and the CODE module feature allows you to implement external source code as a menu command. In addition, QUED/M's macro facility lets you create customized menu commands. We've even improved our acclaimed Find and Replace feature which can search unopened files for literal text or regular expressions created with GREP metacharacters. New features include a file comparison option which combines two or three files into one and marks conflicts automatically. $149

ScriptGen Pro™ by StepUp Software is an Installer script generator which requires no programming or knowledge of Rez. Supports StepUp's InstallerPack, StuffIt compression, custom packages, splash screens, network installs, Rez code output, importing resources, and AppleEvent link w/MPW: $169

SoftPolish by Language Systems is a development tool that helps software developers avoid embarrassing spelling errors, detect incorrect or incompatible resources and improve the appearance of their Macintosh software. SoftPolish examines application resources and reports potential problems to a scrolling log. Independent of any programming language or environment, SoftPolish improves the quality of any Macintosh program. $169

NEW! **Spyer** by InCider is a simple operated tool that records all actions (including mouse movement) you perform on a Macintosh computer and then replays them at your preferred speed. The recorded data can be saved in files for future use. Spyer works as a background process with any Macintosh application and is triggered by user defined Hot Keys. Spyer enables the "Continuous Redo" utility and is especially useful for software testing and demonstration. $39

StoneTable by Stone Tablet Publishing: a library replacing all functions found in list manager plus: variable size columns/rows; different font, size, style, foreground color, background color per cell; sort, resize, move, copy, hide columns/rows; edit cells/titles in place; titles for columns/rows; multiple lines per cell; grid line pattern/color; greater than 32k data per table; up to 32k text per cell; support for balloon help and binary cell data. Versions for THINK C, THINK Pascal, MPW C, MPW Pascal. (all prices per developer) $150, any 2 compilers $200, any 3 compilers $250, all 4 compilers $300

Stone Table Extra: additional functions for StoneTable. Drag selected cells within table or to other tables; optionally add rows as part of drag; popup menus in cells; variable width grid lines; move/drag/resize table in window; clipboard operations on multiple cells. Requires StoneTable. (all prices per developer) $50, any 2 compilers $75, any 3 compilers $100, all 4 compilers $150

ViperBase by Viper Development is a fast database designed for developers that want speed but don't want to spend months or years developing a commercial quality database. ViperBase: Unlimited Records, Variable Length: $59. ViperBase II: ViperBase + Multiple Indices. $119

ADVERTISER/ PRODUCT LIST

LIST OF ADVERTISERS

LIST OF PRODUCTS

TIPS &TIDBITS

By Scott T Boyd, Editor

TIP OF THE MONTH

Faster Color

RGBForeColor and RGBBackColor can take a surprising amount of time, especially if your main drawing loop calls both routines before most drawing operations. Even if you call RGBForeColor with the color that's currently foremost, it still recalculates the best possible foreground color! By remembering the results of RGBForeColor and RGBBackColor, you can significantly increase your drawing speed; this example program shows a drawing speed increase of 20%!

The program is written to be pasted into a brand new Think C Project; no MacTraps library is required here. It dumps you into MacsBug at the end with location $40 holding the unoptimized time and $44 holding the optimized time. You can use locations $40 through $5B, inclusive, for debugging purposes.

```
#include <QuickDraw.h>
#include <Windows.h>
#include <Palettes.h>
#include <Events.h>

void main(void) {
    CWindowRecord   cwr;
    WindowPtr       wp;
    Rect            bounds = {100, 50, 100 + 256, 50 + 100};
    RGBColor        theColor;
    unsigned short  i, lp;
    unsigned short  start, stop, inc;
    long            startT, stopT;
    long            ColorIndex[1000];
    short           optimized;

    InitGraf( NewPtr(2000) + 1000);
    InitCursor();
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(0);

    wp = NewCWindow(&cwr,&bounds,"\pFill",TRUE,zoomDocProc,0,TRUE,237);
    SetPort(wp);
    OffsetRect(&bounds, -bounds.left, -bounds.top);

    for (optimized = 0; optimized <= 1; optimized++) {
        startT = TickCount();

        if (optimized) {
            for (i = 0; i < 256 * 255; i += 256) {
                theColor.red = i;
                theColor.green = i;
                theColor.blue = i;
                RGBForeColor(&theColor);
                ColorIndex[i >> 8] = cwr.port.fgColor;
            }
    }
```

Continued on page 83

Got a developer tip you've been keeping to yourself but really need to share? Think you have a better trick up your sleeve? Send us your tips and tricks, especially programming-related tips, but don't hold back if you've got programmer's user tips.

We want your tips! We pay $25 for every tip used, and $50 for Tip of the Month. You can take your award in orders or subscriptions if you prefer.

Make sure code compiles, and send tips by e-mail. See page two for our addresses.

Not Such A Drag After All

The drag manager is really cool and can make apps a lot more intuitive, but it's a pain to debug since process switches are disabled while drags occur. Since both Think C's and Metrowerks' debugger require these, you cannot use them. Never fear! You can use The Debugger!

While we're on the subject, here's a gotcha for you. Watch out for a bug that causes deadlock if you call WaitNextevent from a drag receive handler.

– Rod Magnuson, Cupertino, CA

Going Faster with Symantec TPM

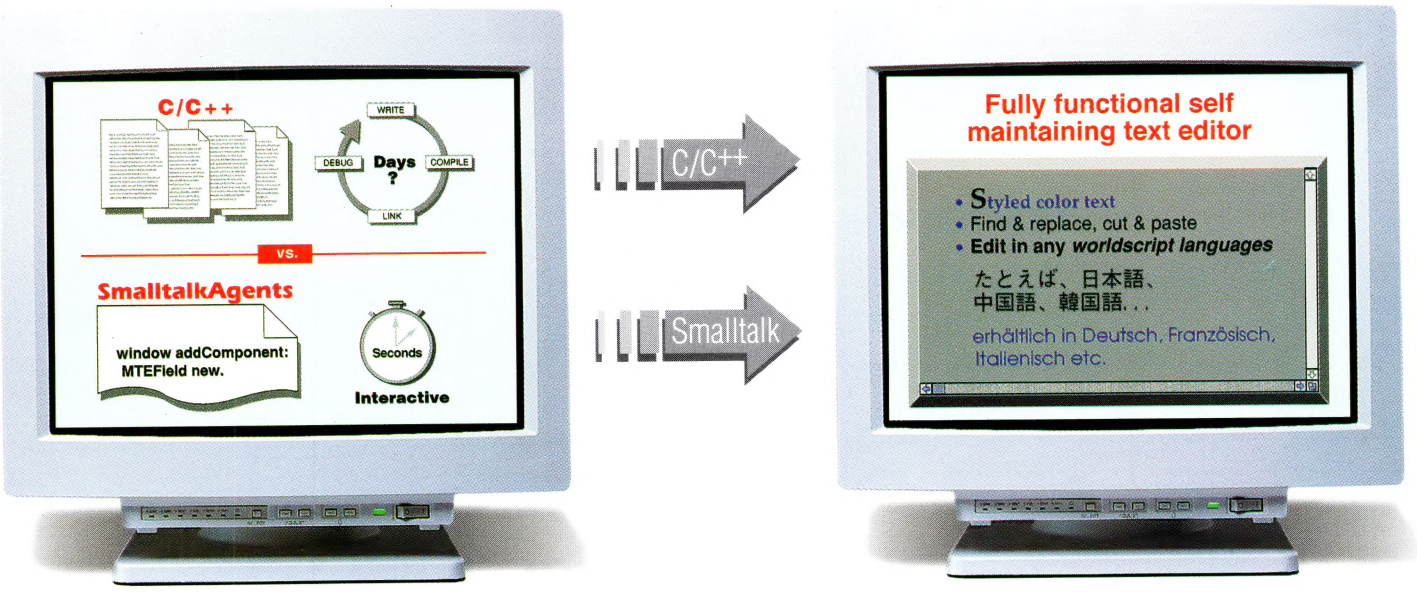
Symantec C++, both versions 6.0 and 7.0, do not have the compilers as part of Think Project Manager. Instead,

Continued on page 83

JOIN THE (R)EVOLUTION

SmalltalkAgents™

An interactive object oriented dynamic language and programming environment.



Build More, Write Less... With SmalltalkAgents' Object Oriented Workbench.

If you have been struggling with C++ object oriented code and libraries, come to the language that *defined* "Object Oriented". A language where the code is understandable and the features are accessible – *not cryptic*. The object oriented concepts are clear, and the tools to navigate, learn, and design are an intrinsic part of the programming environment.

SmalltalkAgents is not *just* another Smalltalk, it is a new generation of Smalltalk, designed for System 7,™ WorldScript,™ and RISC. It is a superset, which combines the results of over ten years of evolution in software and hardware engineering, including key aspects of C and LISP such as structures and list operators.

We have been Apple developers for years, which means that we know Pascal, C/C++ and that we understand the Macintosh® complexities. We designed SmalltalkAgents for the next revolution in desktop computing. And, to make it easier to integrate our OOP technology, we wrote the technical documentation with C and Pascal programmers in mind.

Compatibility – SmalltalkAgents has complete access to the Macintosh Toolbox using standard *Inside Macintosh*® Pascal and Register style calling formats. It provides inline, dynamic trap definition, multi-register return values, intelligent type coercion, callbacks, and trap selectors. Code libraries and resources are accessible through dynamically definable inline foreign function calls.

Reusability – Extensive modular class libraries, that include 32 bit color QuickDraw® user interface components, provide a quality application framework that dramatically reduces development time and complexity.

Productivity – SmalltalkAgents includes interactive cross-reference tools, integrated source code debugger and compilers, and other tools such as object inspectors. SmalltalkAgents scalable approach to building reliable applications enables quick delivery of systems which can be smoothly extended for changing requirements, localization needs of international markets, and migration to future Apple® platforms.

Networking – Pre-emptive interrupt driven threads, dynamically definable Apple Event handlers, AppleTalk® protocol classes, TCP/IP classes, and streams filter classes allows you to cross over into new thresholds of distributed processing, networking tools, server applications.

Test drive our product risk free. We offer a 30-day money-back guarantee. For more information, free product literature, or to place an order, please call today at

1-800-296-1339

QKS

Quasar **K**nowledge **S**ystems • 9818 Parkwood Drive • Bethesda, Maryland 20814-4028 USA • Phone: 301-530-4853 • Fax: 301-530-5712

Looking for a ROBUST COMPACT *FAST* Multi-User Relational Database?

Inside Out II®

"The definitive database engine for the Macintosh. Inside Out's speed is particularly impressive."

David Thompson
Director of Software Development
Sybase DEFT Division

The database powerhouse behind corporate and commercial 3GL development.

Sybase: DEFT®
Ensign Systems: POS•IM™
Aldus Corporation: FETCH®
WestWare: CONTACT EASE™
Multi-Ad Services, Inc.: SEARCH
Blueridge Technologies, Inc.: OPTIX™
Apple Computer, Inc.: APPLE ORDER
Philip Morris: Sales Automation System
US Air Force: Office Automation Systems
Robins Analytics: READY FOR TRIAL!®
Ernst and Young: Audit Automation System
State of the Art Inc: ACCOUNTANT, INC. *Eddy Award Winner*
HealthCare Communications: Medical Office Management Systems

Applications developed and distributed with Inside Out have an annual value in excess of $100 million dollars.

Call 800.621.0631 for FREE information on the developers' choice.

SIERRA SOFTWARE INNOVATIONS

923 Tahoe Blvd.
Incline Village,
Nevada 89451
Phone: 702.832.0300
Fax: 702.832.7753
AppleLink: D2086